L2: Entry 7 of 13 File: USPT May 22, 1990

DOCUMENT-IDENTIFIER: US 4928238 A
TITLE: Scalar data arithmetic control system for vector arithmetic processor

Brief Summary Paragraph Right (7):
A scalar data arithmetic control system for a vector arithmetic processor according to the present invention comprises, in the vector arithmetic processor capable of performing arithmetic operations independently of a CPU under the control of the CPU, a register file for storing an operand when a vector instruction sent from the CPU is executed, an operand pointer register for storing address data for the operand stored in the register file, an operand counter which is incremented when the operand is loaded in the register file and decremented when the vector instruction is executed, and a scalar control register for storing a value representing whether the operand is a scalar or vector operand. The value of the scalar control register is detected as a value for a scalar arithmetic operation, and the same value as that of the first cycle is used in the operand pointer register in the second and subsequent cycles so as to perform scalar arithmetic operations. The CPU does not load the operand in the register file but sets only the operand counter.

Detailed Description Paragraph Right (11):
After the CPU 60 sets the instruction, data of the first operand Ai and data of the second operand Bi are loaded by the CPU 60 through the bus 106 at addresses "8".sub.HEX and "A".sub.HEX of the register file 4, respectively. Upon completion of loading, the operand counter 9 is incremented by one.

Detailed Description Paragraph Right (14):
However, since the register 11 of the scalar control register 1 is set at logic "1", the signal line 103 is set to a value corresponding to "8".sub.HEX. At the time of initiation of the second cycle, values "8".sub.HEX and "E".sub.HEX are respectively set in the pointer registers 2 and 3. Since the operand Ai is scalar data, the CPU 60 uses the data of the first cycle for the operand Ai in the second cycle. The CPU 60 loads only the data of the second cycle for the second operand Bi at address "E".sub.HEX of the register file 4 through the bus 106. The operand counter 9 is incremented by one.

Current US Original Classification (1):
712/7

Current US Cross Reference Classification (2):
712/2

CLAIMS:

1. A scalar data arithmetic control system for a vector arithmetic processor for performing vector arithmetic operations independently of a central processing unit under the control of said central processing unit, said vector arithmetic processor comprising:

a register file for storing an operand supplied from said central processing unit when a vector operation instruction is to be executed;

pointer register means for storing and outputting to said register file address data for said operand stored in said register file;

an operand counter for counting operands supplied from said central processing unit

to said register file, a count of said operand counter being incremented when said operand is loaded in said register file and decremented when said vector operation instruction is executed; and

a scalar control register for storing a value representing whether said operand is a scalar or vector operand, said value being determined by said central processing unit on the basis of said vector operation instruction and supplied therefrom to said scalar control register; said pointer register means being coupled to said scalar control register; wherein

when said value stored in said scalar control register is representative of an operand being scalar, second and subsequent cycles of the arithmetic operations are performed using same address data outputted from said pointer register means for said operand as that used in a first cycle of the arithmetic operations so as to perform scalar arithmetic operations, and said central processing unit sets only said operand counter in place of loading of said operand in said register file.

L2: Entry 8 of 13 File: USPT Dec 5, 1989

DOCUMENT-IDENTIFIER: US 4885678 A
TITLE: Vector processor for processing database without special tags for vector elements

Brief Summary Paragraph Right (11):
A part number of a part name table 701 is stored in a first input operand X and a part number of a price table 702 is stored in a second input operand Y, and the element of X and the element of Y are compared. If both elements are equal, a pair of operand counters of the input operands are generated as tag information to identify both elements and they are outputted as elements of a third output operand Z. The operands X and Y and the counters i and j are initially "1" and the comparison of X(1) and Y(1) is equal (P2=P2). Accordingly, the pair (i, j) or (1, 1) is stored as Z(1). Then, the operand counters i and j are incremented by one and X(2) and Y(2) are compared. Since they are not equal (P3operand counter i is incremented by one, and similar operation is repeated. When the operand counters i and j are 3 and 4, respectively, they are gain equal (P8=P8) and the pair (3, 4) is stored as (Z2).

Detailed Description Paragraph Right (28):
On the other hand, the operand counters OCX, OCY and OCZ are incremented by one by the signals UPX, UPY and UPZ so that they have values "2" (FIG. 3, .circle.9). The operand addresses OAX, OAY and OAZ are also updated to designate the next elements (FIG. 3, .circle.10).

Detailed Description Paragraph Right (29):
As the operand counters and operand addresses are renewed, a second fetch request signal FREQ is sent (FIG. 3, .circle.11). The subsequent process is similar to that for the first FREQ (FIG. 3, .circle.5). Since the comparison result signal CMP now indicates P3P5 (FIG. 3, .circle.14) and only the UPY signal is "1" (FIG. 6, item 2) so that OCY and OAY are updated to "3" and "a2+8), respectively. For a fourth FREQ (FIG. 3, .circle.15), third elements of the vectors X and Y are fetched as shown by OCX and OCY. The fetched data have a relation of P8>P7 and hence the same operation as the previous one (FIG. 6, case 2) is repeated. For a fifth FREQ (FIG. 3, .circle.16), the fetched data are both P8 (FIG. 3, .circle.17) and the signals UPX, UPY, UPZ and SREQ are "1" as they are for the first FREQ, so that the operand counters and address registers are updated, and the contents of OCX and OCY, that is, (3, 4) are written into the main memory 107. The operand counters OCX, OCY and OCZ are incremented by one to "4", "5" and "3", respectively (FIG. 3, .circle.18). The comparison of the contents of OCY and OLY indicates 4>3. This means that the content of the counter exceeds the number of elements. Thus, the control signal END 410 which indicates the termination of execution is set to "1" (FIG. 3, .circle.19), a flip-flop 402 which indicates that the join instruction is under execution is reset, the fetch request FREQ and store request SREQ are stopped and the execution of the join instruction is terminated.

Current US Original Classification (1):
712/6

WEST

Your wildcard search against 2000 terms has yielded the results below

Search for additional matches among the next 2000 terms

Generate Collection | Print

Search Results - Record(s) 1 through 10 of 10 returned.

☐ 1. Document ID: US 6240508 B1

L3: Entry 1 of 10 File: USPT May 29, 2001

DOCUMENT-IDENTIFIER: US 6240508 B1
TITLE: Decode and execution synchronized pipeline processing using decode generated memory read queue with stop entry to allow execution generated memory read

Detailed Description Paragraph Right (71):
Each time valid register mode source specifiers appear on the spec-bus 78 the counters 152 in the source array 150 that correspond with those registers are incremented, as determined by selector 154 receiving the register numbers as part of the information on the bus 78. At the same time, the operand queue unit 79 inserts entries pointing to these registers in the source queue 37. In other words, for each register mode source queue entry, there is a corresponding increment of a counter 152 in the array 150, by the increment control 155. This implies a maximum of two counters incrementing each cycle when a quadword register mode source operand is parsed (each register in the register file 41 is 32-bits, and so a quadword must occupy two registers in the register file 41). Each counter 152 may only be incremented by one. When the execution unit 23 removes the source queue entries the counters 152 are decremented by decrement control 156. The execution unit 23 removes up to two register mode source queue entries per cycle as indicated on the retire bus 82. The GPR numbers for these registers are provided by the execution unit 23 on the retire bus 82 applied to the increment and decrement controllers 155 and 156. A maximum of two counters 152 may decrement each cycle, or any one counter may be decremented by up to two, if both register mode entries being retired point to the same base register.

Current US Original Classification (1):
712/219

Full | Title | Citation | Front | Review | Classification | Date | Reference | Sequences | Attachments | Claims | KWIC | Draw Desc | Image

☐ 2. Document ID: US 6212630 B1

L3: Entry 2 of 10 File: USPT Apr 3, 2001

DOCUMENT-IDENTIFIER: US 6212630 B1
TITLE: Microprocessor for overlapping stack frame allocation with saving of subroutine data into stack area

Detailed Description Paragraph Right (2):

FIG. 5 is a block diagram showing the construction of the processor 10 of the embodiment. The processor 10 handles 32 bits (4 bytes) as its basic word length (one word) and is a microprocessor which sequentially executes instructions with a fixed word length of 32 bits. The processor 10 includes the register file (GR) 11, the stack pointer (SP) 12, the return register (LR) 13, the ALU 14, the program counter (PC) 15, the incrementor (INC) 16, the decoder (Dec) 18, the fixed values ("+4" 20, "-4"21), the adder 22, the internal buses (the A bus 33, the B bus 34, the C bus 35, and the D bus 36), the operand address buffer (OAB) 26, the operand data buffer (ODB) 27, the instruction address buffer (IAB) 28, the instruction data buffer (IDB) 29, and the selectors 30-32. Note that these components are controlled by control signals output from the decoder 18 and operate in synchronization with an internal clock (not shown in FIG. 5). The data memory 38 and the instruction memory 39 are external memories connected to the processor 10 and are respectively a storage area for storing temporary data, such as a stack, and a storage area for prestoring machine instructions.

Current US Original Classification (1):
712/242

Full | Title | Citation | Front | Review | Classification | Date | Reference | Sequences | Attachments | Claims | KWIC | Draw Desc | Image

---

☐ 3. Document ID: US 5809320 A

L3: Entry 3 of 10 File: USPT Sep 15, 1998

DOCUMENT-IDENTIFIER: US 5809320 A
TITLE: High-performance multi-processor having floating point unit

Detailed Description Paragraph Right (102):
Each time valid register mode source specifiers appear on the spec-bus 78 the counters 152 in the source array 150 that correspond with those registers are incremented, as determined by selector 154 receiving the register numbers as part of the information on the bus 78. At the same time, the operand queue unit 79 inserts entries pointing to these registers in the source queue 37. In other words, for each register mode source queue entry, there is a corresponding increment of a counter 152 in the array 150, by the increment control 155. This implies a maximum of two counters incrementing each cycle when a quadword register mode source operand is parsed (each register in the register file 41 is 32-bits, and so a quadword must occupy two registers in the register file 41). Each counter 152 may only be incremented by one. When the execution unit 23 removes the source queue entries the counters 152 are decremented by decrement control 156. The execution unit 23 removes up to two register mode source queue entries per cycle as indicated on the retire bus 82. The GPR numbers for these registers are provided by the execution unit 23 on the retire bus 82 applied to the increment and decrement controllers 155 and 156. A maximum of two counters 152 may decrement each cycle, or any one counter may be decremented by up to two, if both register mode entries being retired point to the same base register.

Current US Original Classification (1):
712/34

Current US Cross Reference Classification (1):
712/42

Full | Title | Citation | Front | Review | Classification | Date | Reference | Sequences | Attachments | KWIC | Draw Desc | Image

---

☐ 4. Document ID: US 5542058 A

L3: Entry 4 of 10 File: USPT Jul 30, 1996

DOCUMENT-IDENTIFIER: US 5542058 A
TITLE: Pipelined computer with operand context queue to simplify context-dependent execution flow

Detailed Description Paragraph Right (101):
Each time valid register mode source specifiers appear on the spec-bus 78 the counters 152 in the source array 150 that correspond with those registers are incremented, as determined by selector 154 receiving the register numbers as part of the information on the bus 78. At the same time, the operand queue unit 79 inserts entries pointing to these registers in the source queue 37. In other words, for each register mode source queue entry, there is a corresponding increment of a counter 152 in the array 150, by the increment control 155. This implies a maximum of two counters incrementing each cycle when a quadword register mode source operand is parsed (each register in the register file 41 is 32-bits, and so a quadword must occupy two registers in the register file 41). Each counter 152 may only be incremented by one. When the execution unit 23 removes the source queue entries the counters 152 are decremented by decrement control 156. The execution unit 23 removes up to two register mode source queue entries per cycle as indicated on the retire bus 82. The GPR numbers for these registers are provided by the execution unit 23 on the retire bus 82 applied to the increment and decrement controllers 155 and 156. A maximum of two counters 152 may decrement each cycle, or any one counter may be decremented by up to two, if both register mode entries being retired point to the same base register.

Current US Cross Reference Classification (1):
712/23

Full | Title | Citation | Front | Review | Classification | Date | Reference | Sequences | Attachments | KWIC | Draw Desc | Image

---

☐ 5. Document ID: US 5488730 A

L3: Entry 5 of 10 File: USPT Jan 30, 1996

DOCUMENT-IDENTIFIER: US 5488730 A
TITLE: Register conflict scoreboard in pipelined computer using pipelined reference counts

Detailed Description Paragraph Right (80):
Each time valid register mode source specifiers appear on the spec-bus 78 the counters 152 in the source array 150 that correspond with those registers are incremented, as determined by selector 154 receiving the register numbers as part of the information on the bus 78. At the same time, the operand queue unit 79 inserts entries pointing to these registers in the source queue 37. In other words, for each register mode source queue entry, there is a corresponding increment of a counter 152 in the array 150, by the increment control 155. This implies a maximum of two counters incrementing each cycle when a quadword register mode source operand is parsed (each register in the register file 41 is 32-bits, and so a quadword must occupy two registers in the register file 41). Each counter 152 may only be incremented by one. When the execution unit 23 removes the source queue entries the counters 152 are decremented by decrement control 156. The execution unit 23 removes up to two register mode source queue entries per cycle as indicated on the retire bus 82. The GPR numbers for these registers are provided by the execution unit 23 on the retire bus 82 applied to the increment and decrement controllers 155 and 156. A

maximum of two counters 152 may decrement each cycle, or any one counter may be decremented by up to two, if both register mode entries being retired point to the same base register.

Current US Original Classification (1):
712/41

Current US Cross Reference Classification (1):
712/217

Full | Title | Citation | Front | Review | Classification | Date | Reference | Sequences | Attachments | KWIC | Draw Desc | Image

---

☐ 6. Document ID: US 5471591 A

L3: Entry 6 of 10 File: USPT Nov 28, 1995

DOCUMENT-IDENTIFIER: US 5471591 A
TITLE: Combined write-operand queue and read-after-write dependency scoreboard

Detailed Description Paragraph Right (73):
Each time valid register mode source specifiers appear on the spec-bus 78 the counters 152 in the source array 150 that correspond with those registers are incremented, as determined by selector 154 receiving the register numbers as part of the information on the bus 78. At the same time, the operand queue unit 79 inserts entries pointing to these registers in the source queue 37. In other words, for each register mode source queue entry, there is a corresponding increment of a counter 152 in the array 150, by the increment control 155. This implies a maximum of two counters incrementing each cycle when a quadword register mode source operand is parsed (each register in the register file 41 is 32-bits, and so a quadword must occupy two registers in the register file 41). Each counter 152 may only be incremented by one. When the execution unit 23 removes the source queue entries the counters 152 are decremented by decrement control 156. The execution unit 23 removes up to two register mode source queue entries per cycle as indicated on the retire bus 82. The GPR numbers for these registers are provided by the execution unit 23 on the retire bus 82 applied to the increment and decrement controllers 155 and 156. A maximum of two counters 152 may decrement each cycle, or any one counter may be decremented by up to two, if both register mode entries being retired point to the same base register.

Current US Original Classification (1):
712/217

Full | Title | Citation | Front | Review | Classification | Date | Reference | Sequences | Attachments | KWIC | Draw Desc | Image

---

☐ 7. Document ID: US 5450555 A

L3: Entry 7 of 10 File: USPT Sep 12, 1995

DOCUMENT-IDENTIFIER: US 5450555 A
TITLE: Register logging in pipelined computer using register log queue of register content changes and base queue of register log queue pointers for respective instructions

Detailed Description Paragraph Right (103):

Each time valid register mode source specifiers appear on the spec-bus 78 the counters 152 in the source array 150 that correspond with those registers are incremented, as determined by selector 154 receiving the register numbers as part of the information on the bus 78. At the same time, the operand queue unit 79 inserts entries pointing to these registers in the source queue 37. In other words, for each register mode source queue entry, there is a corresponding increment of a counter 152 in the array 150, by the increment control 155. This implies a maximum of two counters incrementing each cycle when a quadword register mode source operand is parsed (each register in the register file 41 is 32-bits, and so a quadword must occupy two registers in the register file 41). Each counter 152 may only be incremented by one. When the execution unit 23 removes the source queue entries the counters 152 are decremented by decrement control 156. The execution unit 23 removes up to two register mode source queue entries per cycle as indicated on the retire bus 82. The GPR numbers for these registers are provided by the execution unit 23 on the retire bus 82 applied to the increment and decrement controllers 155 and 156. A maximum of two counters 152 may decrement each cycle, or any one counter may be decremented by up to two, if both register mode entries being retired point to the same base register.

Current US Original Classification (1):
712/228

Current US Cross Reference Classification (1):
712/218

Full | Title | Citation | Front | Review | Classification | Date | Reference | Sequences | Attachments KWIC | Draw Desc | Image

---

☐ 8. Document ID: US 5394529 A

L3: Entry 8 of 10 File: USPT Feb 28, 1995

DOCUMENT-IDENTIFIER: US 5394529 A
TITLE: Branch prediction unit for high-performance processor

Detailed Description Paragraph Right (72):
Each time valid register mode source specifiers appear on the spec-bus 78 the counters 152 in the source array 150 that correspond with those registers are incremented, as deter:mined by selector 154 receiving the register numbers as part of the information on the bus 78. At the same time, the operand queue unit 79 inserts entries pointing to these registers in the source queue 37. In other words, for each register mode source queue entry, there is a corresponding increment of a counter 152 in the array 150, by the increment control 155. This implies a maximum of two counters incrementing each cycle when a quadword register mode source operand is parsed (each register in the register file 41 is 32-bits, and so a quadword must occupy two registers in the register file 41). Each counter 152 may only be incremented by one. When the execution unit 23 removes the source queue entries the counters 152 are decremented by decrement control 156. The execution unit 23 removes up to two register mode source queue entries per cycle as indicated on the retire bus 82. The GPR numbers for these registers are provided by the execution unit 23 on the retire bus 82 applied to the increment and decrement controllers 155 and 156. A maximum of two counters 152 may decrement each cycle, or any one counter may be decremented by up to two, if both register mode entries being retired point to the same base register.

Current US Original Classification (1):
712/240

Full | Title | Citation | Front | Review | Classification | Date | Reference | Sequences | Attachments KWIC | Draw Desc | Image

☐ 9. Document ID: US 4992934 A

L3: Entry 9 of 10 File: USPT Feb 12, 1991

DOCUMENT-IDENTIFIER: US 4992934 A
TITLE: Reduced instruction set computing apparatus and methods

Brief Summary Paragraph Right (37):
The hardwired control unit decodes instruction signals for: (i) addressing sequentially stored instructions by providing an incrementing signal to the RISC instruction counter during a first quarter of each machine cycle; (ii) providing, during a second quarter of selected machine cycles, control signals to a register file for selecting storage registers in the register file to have their signal contents operated on by the ALU and for providing, during the second quarter, the selected register signal contents for storage in the source and destination registers; (iii) providing, during a third quarter of selected machine cycles, enabling signals for enabling the operand address and data buses in response to an instruction to load or store data from memory to the register file or to memory from the register file; (iv) providing, starting during a third quarter of selected machine cycles, a first select signal to the AMUX for selecting between the source output signal and the instruction signal for provision as said first input signal for the ALU; (v) providing, starting during the third quarter of selected machine cycles, a second select signal to the BMUX for selecting between the destination output signal, the accumulator output signal and the instruction address signal for provision as the second input signal for the ALU; (vi) selecting, starting during the second quarter of selected machine cycles, an operation to be performed by the ALU by providing an ALU operation select signal to the ALU; (vii) storing, during a first quarter of selected machine cycles, the ALU output signal in the register file, the accumulator, or the instruction counter by providing an ALU output select signal to the appropriate register; and (viii) providing shift signals, during an extended fourth quarter of selected machine cycles, for performing shift, multiplication and division operations.

Detailed Description Paragraph Right (15):
Thusfar, the control unit 30 has been described performing its functions of fetching RISC instructions by incrementing the instruction counter 32, decoding current instructions received in both the primary ("pipe") register and secondary register of an instruction register 34 for performing various control functions internally within the RISC machine 10, including selecting registers to be operated on within the register file 74 and storing them into source and destination registers 76, 78, and loading either an ALU output signal from the previous instruction into a selected register in the register file 74 or loading an operand from the data bus 18 if the decoded operand indicates a load operation from memory.

Current US Original Classification (1):
712/209

Full | Title | Citation | Front | Review | Classification | Date | Reference | Sequences | Attachments | KWIC | Draw Desc | Image

☐ 10. Document ID: US 4928238 A

L3: Entry 10 of 10 File: USPT May 22, 1990

DOCUMENT-IDENTIFIER: US 4928238 A
TITLE: Scalar data arithmetic control system for vector arithmetic processor

Brief Summary Paragraph Right (7):
A scalar data arithmetic control system for a vector arithmetic processor according to the present invention comprises, in the vector arithmetic processor capable of performing arithmetic operations independently of a CPU under the control of the CPU, a register file for storing an operand when a vector instruction sent from the CPU is executed, an operand pointer register for storing address data for the operand stored in the register file, an operand counter which is incremented when the operand is loaded in the register file and decremented when the vector instruction is executed, and a scalar control register for storing a value representing whether the operand is a scalar or vector operand. The value of the scalar control register is detected as a value for a scalar arithmetic operation, and the same value as that of the first cycle is used in the operand pointer register in the second and subsequent cycles so as to perform scalar arithmetic operations. The CPU does not load the operand in the register file but sets only the operand counter.

Current US Original Classification (1):
712/7

Current US Cross Reference Classification (2):
712/2

CLAIMS:

1. A scalar data arithmetic control system for a vector arithmetic processor for performing vector arithmetic operations independently of a central processing unit under the control of said central processing unit, said vector arithmetic processor comprising:

a register file for storing an operand supplied from said central processing unit when a vector operation instruction is to be executed;

pointer register means for storing and outputting to said register file address data for said operand stored in said register file;

an operand counter for counting operands supplied from said central processing unit to said register file, a count of said operand counter being incremented when said operand is loaded in said register file and decremented when said vector operation instruction is executed; and

a scalar control register for storing a value representing whether said operand is a scalar or vector operand, said value being determined by said central processing unit on the basis of said vector operation instruction and supplied therefrom to said scalar control register; said pointer register means being coupled to said scalar control register; wherein

when said value stored in said scalar control register is representative of an operand being scalar, second and subsequent cycles of the arithmetic operations are performed using same address data outputted from said pointer register means for said operand as that used in a first cycle of the arithmetic operations so as to perform scalar arithmetic operations, and said central processing unit sets only said operand counter in place of loading of said operand in said register file.

Term	Documents
REGISTER.USPT.	175781
REGISTERS.USPT.	99809
FILE.USPT.	86480
FILES.USPT.	38761
COUNTER.USPT.	322688
COUNTERS.USPT.	53163
OPERAND$	0
OPERAND.USPT.	10052
OPERANDA.USPT.	4
OPERANDADDRESS/RAM.USPT.	1
OPERANDAND.USPT.	1
(L1 AND ((REGISTER ADJ FILE) WITH COUNTER WITH OPERAND$ WITH INCREMENT$)).USPT.	10

There are more results than shown above. Click here to view the entire set.

Display Format: KWIC Change Format

Previous Page Next Page

L2: Entry 11 of 13 File: USPT Apr 25, 1978

DOCUMENT-IDENTIFIER: US 4086626 A
TITLE: Microprocessor system

Detailed Description Paragraph Right (98):
The program counter 42 contains the address of the next instruction code to be retrieved from the memory within the ROM 22. The instruction code retrieved from the memory is an 8-bit byte, and once this code has been retrieved from the memory, the program counter 42 is automatically incremented. However, there are other ways of modifying the contents of the program counter 42, which will be explained in greater detail hereinbelow. The data counter 527 is employed for referencing memory addresses. A select group of instruction codes are employed with the microprocessor system which use the data counter 527 to address, or point to, their operands in the memory space. The data counter 527, like the program counter 42, is incremented by one at the conclusion of a memory addressing circuit. Thus the data counter will be pointing to the next location in memory after execution of the memory addressing cycle. The data counter operates independent of the program counter 42 and thereby can address a group of cells in the memory which are wholly separate or independent of those groups of cells addressed by the program counter 42.

Detailed Description Paragraph Right (108):
The 16 bit address, which was supplied to the address gating circuit 532, is also supplied to the incrementer adder 537 by means of the lines 535. The incrementer adder 537 performs the function of incrementing the address value by "1" at the completion of an address cycle, and transmits this incremented value as a new address back to either the program counter 42 or the data counter 527 by means of the lines 538 and the address transfer bus 522. Thus, the program counter 42 and the data counter 527 are in effect registers, while the incrementing (adding) operation is performed by the incrementer adder 537.

Detailed Description Paragraph Right (110):
An address having some finite binary value may be supplied to the program counter 42 from other sources. For example, an address value may be generated in the CPU 20. This address value may be stored in the RAM 33 in the form of two 8-bit bytes. These two 8-bit bytes are transmitted by means of the lines 34 to the internal data bus 500 of the ROM circuit 22, and to the input of the address multiplexer circuit 516 by means of the lines 517. The first byte applied at the input of the circuit 516 is supplied to the upper 8-bit positions of the 16-bit position address transfer bus 522 by means of the lines 520. The second 8-bit byte applied at the input of the circuit 516 is supplied to the lower 8-bit positions of the address transfer bus 522 by means of the line 521. Thus, the two 8-bit bytes transmitted from the CPU 20 are assembled in the program counter 42 or the data counter 527. The subsequent address cycles of the ROM storage 548 is the same as that described hereinabove and the addressing counters (i.e., program counter 42 or data counter 527) are sequentially incremented from the newly assembled address value by means of the incrementer adder 537.

Current US Original Classification (1):
712/244

Current US Cross Reference Classification (2):
712/212

WEST

Generate Collection

L11: Entry 1 of 3 File: USPT Oct 30, 2001

DOCUMENT-IDENTIFIER: US 6311261 B1
TITLE: Apparatus and method for improving superscalar processors

Brief Summary Paragraph Right (7):
First, multiple scalar instructions are fetched simultaneously from an instruction cache/memory or other storage unit. Current state-of-the-art superscalar microprocessors fetch two or four instructions simultaneously. Valid fetched instructions (the ones that are not after a branch-taken instruction) are decoded concurrently, and dispatched into a central instruction window (FIG. 1a) or distributed instruction queues or windows (FIG. 1b). Shelving of these instructions is necessary because some instructions cannot execute immediately, and must wait until their data dependencies and/or resource conflicts are resolved. After an instruction is ready it is issued to the appropriate functional unit. Multiple ready instructions are issued simultaneously, achieving parallel execution within the processor. Execution results are written back to a result buffer first. Because instructions can complete out-of-order and speculatively, results must be retired to register file(s) in the original, sequential program order. An instruction and its result can retire safely if it completes without an exception and there are no exceptions or unresolved conditional branches in the preceding instructions. Memory stores wait at a store buffer until they can commit safely.

Brief Summary Paragraph Right (8):
The parallel executions in superscalar processors demand high memory bandwidth for instructions and data. Efficient instruction bandwidth can be achieved by aligning and merging the decode group. Branching causes wasted decoder slots on the left side (due to unaligned branch target addresses) and on the right side (due to a branch-taken instruction that is not at the end slot). Aligning shifts branch target instructions to the left most slot to utilize all decoder slots. Merging fills the slots to the right of a branch-taken instruction with the branch target instructions, combining different instruction runs into one dynamic instruction stream. Efficient data bandwidth can be achieved by load bypassing and load forwarding (M. Johnson, Superscalar Microprocessor Design, Prentice-Hall, 1991), a relaxed or weak-memory ordering model. Relaxed ordering allows an out-of-order sequence of reads and writes, to optimize the use of the data bus. Stores to memory cannot commit until they are safe (retire step). Forcing loads and stores to commence in order will delay the loads significantly and stall other instructions that wait on the load data. Load bypassing allows a load to bypass stores in front of it (out-of-order execution), provided there is no read-after-write hazard. Load forwarding allows a load to be satisfied directly from the store buffer when there is a read-after-write dependency. Executing loads early is safe because load data is not written directly to the register file.

Brief Summary Paragraph Right (45):
These and other objects, advantages, and features are accomplished by the provision of a method of using a modified reorder buffer in a processor, comprising the steps of: allocating a plurality of entries to the modified reorder buffer for each of a plurality of register assignments during a decode stage of the processor in a cycle of a clock associated with said processor; presenting a plurality of unique operand tags to the modified reorder buffer to read a corresponding plurality of register values during an issue/execute stage of the processor; writing result values to the register value fields of previously allocated modified reorder buffer entries during a writeback stage of the processor; and checking to see if all or some of predetermined ones of the plurality of previously allocated modified reorder buffer entries can retire to a register file during a retire stage of the processor.

Brief Summary Paragraph Right (46):
These and other objects, advantages, and features are provided by a processor, comprising: a fetch and decode unit to receive a first information from at least one storage device; a plurality of functional units connected to the fetch and decoder unit by a global bus; at least one register unit having a modified reorder buffer connected to a corresponding at least one register file by a retire bus, the at least one register unit connected to the plurality of functional units by a local bus, wherein the modified reorder buffer retires a second information related to the first information to the at least one register file over the retire bus, and the at least one register unit receives a third information related to the first and second information from the plurality of functional units.

Brief Summary Paragraph Right (47):
These and other objects, advantages, and features are provided by a processor, comprising: a fetch and decode unit for fetching and decoding information; a plurality of functional units connected to the fetch and decode unit by a global bus; and a plurality of distributed instruction queues, each of the plurality of distributed instruction queues located in a corresponding one of the functional units for receiving instructions from the fetch and decode unit over the global bus.

Brief Summary Paragraph Right (48):
These and other objects, advantages, and features are provided by a processor, comprising: at least one register unit disposed in the processor having registers; a register file contained in each of the at least one register unit; and a corresponding modified reorder buffer contained in each of the at least one register unit connected to the register file, wherein the corresponding modified reorder buffer supports register renaming, out-of-order execution, multi-level speculative execution, and precise interrupts.

Brief Summary Paragraph Left (1):
To improve performance (reduce execution time), it is necessary to reduce one or more factors. The obvious one to reduce is Clock_Period, by means of semiconductor/VLSI technology improvements such as device scaling, faster circuit structures, better routing techniques, etc. A second approach to performance improvement is architecture design. CISC and VLIW architectures take the approach of reducing Inst_Count. RISC and superscalar architectures attempt to reduce the CPI. Superpipelined architectures increase the degree of pipelining to reduce the Clock_Period.

Drawing Description Paragraph Right (2):
FIG. 2 shows a register mapping table in the IBM RS/6000 floating point unit.

Drawing Description Paragraph Right (33):
FIG. 33 illustrates a floating-point arithmetic logic unit (FALU).

Detailed Description Paragraph Right (4):
Result shelving is a technique to temporarily shelve register results before they can be safely retired to a register file. This usually involves buffering of multiple updates to the same register, thus allowing multiple copies/instances of a register to exist in a processor. Four result shelving techniques are discussed in the following sections: register scoreboarding, register-mapping table, checkpoint repair, and reorder buffer. These result shelving techniques are used to support certain superscalar features such as out-of-order execution, register renaming, speculative execution, and precise interrupts.

Detailed Description Paragraph Right (5):
To support the above superscalar features, the complete processor state, which includes not only register file(s) but also software-visible memory, should be shelved. (It is assumed that a processor's register file(s) contains all architectural registers, including status/control registers.) Stores to the memory can be shelved separately into a store buffer. Each store instruction in the store buffer is committed only if it is safe, meaning: (a) the store instruction can be executed without an exception error (page fault, illegal address), (b) instructions prior to the store instruction are also exception-free, and (c) prior conditional

branches have been resolved. These conditions guarantee in-order, non-speculative stores. Thus, despite possible out-of-order and speculative executions inside the processor, the memory is guaranteed to contain the same state as if the processor executed the code in the original, serial, sequential program order.

Detailed Description Paragraph Right (6):
The register scoreboarding technique was originally introduced by Thornton in the CDC 6600 (see J. E. Thornton, Design of a Computer-The Control Data 6600, Scott, Foresman and Co., 1970, M. Johnson, Superscalar Microprocessor Design, Prentice-Hall, 1991). It is the simplest form of result shelving. There is only one level of result shelving, which is accomplished by copying operands and forwarding results to an instruction window. It only allows a maximum of one pending update to a register. Each register in the register file is associated with a scoreboard bit. A "0" indicates the register content is valid, and a "1" indicates there is a pending update. When a register is updated/written, its scoreboard bit is reset. The term "scoreboarding" used in typical commercial microprocessors, such as the Motorola 88100 and Intel i960CA, does not fit Thornton's definition because they are only used to detect and enforce dependencies (see Johnson, 1991). Instead of buffering to an instruction window, these microprocessors simply stall instruction decoding when data dependencies and resource conflicts are detected.

Detailed Description Paragraph Right (12):
The floating-point unit of the IBM RS/6000 (Trademark of IBM, Inc.) uses a 32-entry, 6-bit-wide register mapping table to implement register renaming, as shown in FIG. 2. There are 32 architectural registers and 40 physical registers. Some parts of the register-renaming structure are intentionally omitted from the original diagram to focus on the register mapping table. This register-renaming structure is implemented as a solution to the out-of-order completion problem of floating-point load/store and floating-point arithmetic instructions. In the IBM RS/6000, floating-point load/store instructions are performed independently at the fixed-point unit, which involve load/store address calculations. Without renaming, out-of-order completion can violate anti or output dependency. For example, in a floating-point load operation, the load data may return too early and overwrite a register that is still needed (i.e., it has not been read by an earlier floating-point arithmetic instruction).

Detailed Description Paragraph Right (13):
The register renaming process is done as follows. For notational purpose, MP(i)=j (the contents of the mapping table at address i is j) indicates that architectural register Ri maps to physical register Rj. Initially, the mapping table (MP) is reset to identity mapping, MP(i)=i for i=0, . . . , 31. A remapping is performed for every floating-point load/store instruction decoded. Suppose a floating-point load to architectural register 3, FLD R3, arrives at MP. First, the old entry of MP(3), i.e., index 3, is pushed onto the pending-target return queue (PTRQ). Then, a new physical register index from the free list (FL), index 32, is entered to MP(3). This means R3 is now remapped to R32. Further instruction codes with source operand R3 will automatically be changed to R32. Index 3 in the PTRQ is returned to FL (for reuse) only when the last arithmetic or store instruction referencing R3, prior to the FLD R3 instruction, has been performed. This ensures that the current value in physical register R3 is not overwritten while still being used or referenced.

Detailed Description Paragraph Right (15):
The reorder buffer (RB) is a content-addressable, circular buffer with head and tail pointers. Entries are pushed in (allocate) and popped out (retire) in a FIFO (first-in-first-out) manner. It was originally proposed by Smith and Plezkun (see J. E. Smith and A. R. Pleszkun, "Implementation of Precise Interrupts in Pipelined Processors," Proceedings of the 12th Annual Symposium on Computer Architecture, pp. 36-44, 1985) as a hardware scheme to support precise interrupts in pipelined, scalar processors with out-of-order completion. FIG. 3 shows the RB organization. The RB contains the processor's look-ahead state, while the register file (RF) contains the in-order state. The result shift register (RSR) is used to control/reserve the single result bus. (In Smith and Plezkun's processor model, multiple functional units share one result bus.) When an instruction is issued, it reserves stage i of the RSR, where i is the instruction latency (in cycles). If stage i is already reserved, the instruction issue is stalled. The RSR shifts one position every cycle

(toward a smaller stage number). When a valid RSR entry reaches stage 1, the result bus control is set such that in the next cycle the result from the entry's functional unit is gated to the RB.

Detailed Description Paragraph Right (17):
Smith and Plezkun state that the RB'S allocate operation is performed when an instruction is issued for execution to a functional unit. However, this will restrict the RB to support only out-of-order completion with in-order-issue. To also support out-of-order issue (full or partial), it is necessary to allocate entry when an instruction is decoded. This guarantees that instructions are allocated RB entries in the original program order. For example, FIG. 3 shows the contents of the RB after the RB allocations of I6 and I7. When I7 is decoded, an entry is allocated at the tail of RB (then at entry number 5), following I6's entry. The allocated entry is initialized with "dest. reg."=0 and "program counter"=7. The "valid" bit is also reset to indicate that the R0 value is being computed. The tail pointer is then incremented, modulo RB size. FIG. 3 also shows the RSR contents after I7 is issued (a cycle after I6 is issued). An entry is entered at stage 2 to reserve the result bus, because an integer add takes 2 cycles to complete. The previously allocated RB entry number/tag (5) is written to the RSR entry. The valid bit is set to validate entry.

Detailed Description Paragraph Right (24):
Table 1 summarizes result shelving techniques, their features and shortcomings. Review of the literature suggests that the reorder buffer (RB) is the most complete result shelving technique (see Table 1). The closest contender to the RB is the register-mapping table, (also called "rename buffer", implemented as a multi-ported lookup table array), which is used in IBM POWERPCs, MIPS R10000 (Trademark of MIPS Technologies, Inc.), and HP PA-8000 (Trademark of Hewlett-Packard, Inc.). The register-mapping table technique has four disadvantages compared to the RB. First, to read a register operand, it has to access the mapping table, using the logical register number, to get the corresponding physical register number in the register file. This additional delay could potentially lengthen the processor's cycle time or introduce another pipeline stage. The third pipeline stage in the MIPS R10000 is dedicated solely to read operand registers.

Detailed Description Paragraph Right (25):
Second, the mapping table is not a small circuit. For instance, the MIPS R10000 requires two 32.times.6 mapping tables, one with 12 read ports and 4 write ports for the integer register map, and another with 16 read ports and 4 write ports for the floating-point register map.

Detailed Description Paragraph Right (30):
The dispatch stack (DS) is a central instruction window that performs dynamic code scheduling on the dynamic instruction stream of multiple functional unit processors. It allows out-of-order, multi-instruction issue. The instruction window behaves like a stack where instructions are allocated at the top and issued from the bottom. After a set of instructions is issued, the gaps (freed entries) are filled with unissued instructions above it (compression). Then, the next set of instructions can be pushed in. This is important to determine instruction order during dependency checks. A DS entry consists of an instruction tag, opcode, source and destination register identifiers, dependence fields, and issue index. To explain how the DS works, consider the program example shown below (see R. D. Acosta, J. Kjelstrup, and H. C. Torng, "An Instruction Issuing Approach to Enhancing Performance in Multiple Functional Unit Processors," IEEE Transactions on Computers, Vol. C-35, pp. 815-828, 1986), which adds floating-point numbers in R0 through R7 and leaves the sum in R0.

Detailed Description Paragraph Right (34):
To allow multiple instances of a register (register renaming), each register in the register file is associated with two counters (NI and LI). The NI counter represents the number of instances of a register in the RUU, while the LI counter represents the latest instance number. When an instruction with destination register Ri is decoded and allocated to the RUU, Ri's NI and LI counters are incremented. Wrap around in the LI counter is allowed (modulo counter). However, when the NI counter reaches its maximum value, the instruction decoding is stalled. When an instruction is retired from the RUU and updates the destination register, the NI counter is

decremented. With these counters, tag allocations and deallocations become simple. A register tag now simply consists of the register number appended to the LI counter.

Detailed Description Paragraph Right (35):
In each clock cycle, the RUU performs four operations simultaneously: (a) dispatch/allocate one instruction from the decoder, (b) issue one instruction nearest to the head pointer with ready operands, (c) writeback any result value to the instruction's RUU entry, and forward this result to any matching operand(s), and (d) retire one completed instruction at the head entry and update its result value to the register file. To describe these operations, consider the previous program listing. FIG. 5(a) shows the instruction timing when each instruction is allocated, issued, written back, and retired. It is assumed that each floating-point add takes six cycles to complete. FIG. 5(b) shows the snapshot of the RUU contents at cycle 7. Instruction I6 (fadd R0,R4,R0) has just been allocated at the tail of the RUU. Its program counter, functional unit source, operands, and destination register tag are written. The destination register tag (0,3) is simply the register number (0) appended with the current LI counter value for R0 (3). The "executed" flag is reset to indicate entry 6 as unissued. Operands are read directly from the register file. If an operand is available, its value is copied to the allocated entry and the "ready" flag is set. However, if an operand has not been produced, then its register tag is copied to the allocated entry and the "ready" flag is reset. Later when the operand value is produced, the RUU forwards it. By copying operands and forwarding results to the instruction window, anti and output dependencies are effectively eliminated (see Johnson, 1991).

Detailed Description Paragraph Right (38):
The retire operation is performed on the instruction at the bottom entry of the RUU. If its "executed" flag=1, the result value (destination's "content") is retired to the register file and the head pointer is incremented, modulo RUU size (see I0 in FIG. 5(c)). Retiring from the bottom entry ensures that the register file is updated in the original program order. If the instruction does not complete successfully due to exception or branch misprediction, then a recovery action is initiated by flushing the entire contents of RUU. The register file automatically contains the correct in-order state at the branch point or precise-repair point. The NI and LI counters are reset to zero since the only instance of a register is in the register file (register instances in RUU are completely flushed).

Detailed Description Paragraph Right (42):
There are four stages/operations in the Metaflow architecture involving the DRIS: dispatch/allocate, issue, writeback, and retire. (To be consistent with the terms used in this document, the Metaflow's terms of "issue", "schedule", and "update" have been changed to the similar terms dispatch/allocate, issue, and writeback, respectively.) To describe these operations, consider the previous program example. FIG. 7(a) shows the instruction timing when each instruction is allocated, issued, written back, and retired. It is assumed that there are four allocate ports, four retire ports, and two floating-point adders with 3-cycle latency. FIGS. 7(b) and 7(c) show the DRIS contents at different time points.

Detailed Description Paragraph Right (43):
The allocate operation is performed when a set of decoded instructions arrives at the DRIS. For each instruction, the allocated DRIS entry is written with the program counter, opcode, functional unit class number, register numbers of source and destination operand(s), register tags of source operand(s), and some control information. FIG. 7(b) shows a snapshot of the DRIS at cycle 2 when I4, I5, and I6 have just been allocated. The "dispatched" and "executed" bits are initially reset. There are three associative searches involved for each entry allocation; during the setting of "latest" bit in the destination section and during the setting of "locked" bit and "ID" fields in both source operand sections (dependency checking). Consider next the allocation of I6. To determine the "latest" bit, the DRIS allocate logic searches and compares all other entries (including the ones currently being allocated) with the same destination register number as I6 (0). If no match, the "latest" bit is automatically set. In this case, there are two matched entries (I0 and I2). The ID (color, index) comparisons of I0, I2, and I6 show that I6's entry is the youngest. This means I6 is the last instruction in the DRIS that updates R0. So, I6's "latest" bit is set and I2's and I0's "latest" bits are reset. To determine the

"locked" bit and "ID" fields of 16's second operand (R4), register number 4 is associatively searched among the destination registers of other older entries. If no match is found, no instructions in the DRIS updates R4 and R4 must be retrieved from the register file. The "locked" bit is cleared and the "ID" field is set to a default value (e.g. the instruction's own ID) to indicate that the operand is in the register file. However, in this case, two matches are found (I3 and I5), but IS's entry contains the latest R4. Therefore, its index (5) plus the current color bit (0) becomes the ID (0,5) of I6's second operand. The "locked" bit and "ID" fields of I6's first operand are determined similarly.

Detailed Description Paragraph Right (44):
The issue operation is performed by checking the "locked" bits of unissued instructions ("dispatched"=0). The oldest instruction with both operands unlocked ("locked"=0) is given priority for issue, provided the requisite functional unit is available. The DRIS issue logic strives to find as many instructions to issue as possible. FIG. 7(b) shows a case where the DRIS finds two instructions (I0 and I1) to issue. The DRIS issue logic checks the ID field of each source operand to determine the operand location (either in the register file or the DRIS). Since I0's and I1's source operand IDs are the instruction's own ID (default value for register file location), the operand values are fetched from the register file. In the case that an operand is in the DRIS, the operand ID's index part locates the DRIS entry, where the "content" field of the destination section is then routed as the operand value. In any case, the DRIS issue logic passes the opcode, operand values, and ID of each issued instruction to the corresponding functional unit. The "dispatched" bit of the issued entry is then set.

Detailed Description Paragraph Right (46):
The retire operation is performed in order, starting from the oldest instruction in the DRIS (bottom entry). Multiple instructions may be retired simultaneously depending on the number of retired ports and the eligibility of instructions. An instruction is eligible to retire if (see Popescu, et al.): (a) it has completed successfully ("executed"=1 and no exception error), (b) all older instructions have been retired or are being retired in this clock cycle, and (c) there is an available retire port to write the register "content" to the register file. FIG. 7(c) shows two instructions (I0 and I1) are eligible to retire.

Detailed Description Paragraph Right (49):
Compared to the register update unit, the DRIS improves three areas (see Popescu, et al.): (a) register renaming, (b) operand storage and routing, and (c) branch misprediction delay. First, the implementation of register renaming is improved by a more efficient register tagging (ID). Costly counters per register are avoided. The LI and NI counters also limit the number of instances of a register, which could result in more stalling. Second, the storage per entry in the DRIS is less than the RUU because operand values are not copied to DRIS entries. Rather, the operand values are read directly from the DRIS destination section (or the register file) at issue time. This strategy also saves expensive data routing during result forwarding, eliminating paths from result values to operand fields. Third, branch misprediction is repaired as soon as it is issued and executed, not when it is retired at the bottom of the window.

Detailed Description Paragraph Right (51):
Tomasulo introduced reservation stations in the floating-point section of the IBM 360/91 to exploit the multiple execution units. The main objective was to permit simultaneous execution of independent instructions while preserving the precedence (data dependency) constraints in the instruction stream. Reservation stations are essentially an implementation of distributed instruction windows with some result shelving. The result shelving, done by copying operands and result forwarding to reservation stations, are necessary to support register renaming. FIG. 8 shows the hardware implementation of Tomasulo's algorithm. (To focus the discussion on the reservation stations, two hardware units (floating point buffers and store data buffers) are intentionally omitted from the original diagram.)

Detailed Description Paragraph Right (52):
There are four key components in Tomasulo's concept: busy bit, tag, reservation station, and common data bus. A busy bit is associated with each floating-point

register or operand as a dependency mechanism. If set (busy bit=1) then it means the register is not available, currently being generated. A tag is associated with each register instance, which is used in place of the register number/identifier. This reintroduces the distinction between register and value, the essence of register renaming. In Tomasulo' algorithm, a tag corresponds directly (1-to-1) to a reservation station. For example, in the IBM 360/91, tag numbers 10 through 12 correspond to the three reservation stations of the adder unit. However, Weiss and Smith suggested a more flexible way of assigning tags (S. Weiss and J. E. Smith, "Instruction Issue Logic in Pipelined Supercomputers," IERE Transactions on Computers, Vol. C-33, No. 11, pp. 1013-1022, 1984). When an instruction is decoded, a new tag is assigned for its destination register from a "tag pool" that consists of some finite set of tags. When the instruction completes, the tag is returned to the pool for reuse.

Detailed Description Paragraph Right (54):
The actions taken during instruction decode are as follows. The decoder decodes one instruction from the top of the floating point operation stack (FLOS). A reservation station is allocated at the appropriate execution unit. Instruction operands (sink and source) are copied from FLR to the reservation stations. If the busy bit of an operand register in FLR is clear (indicating the register value is valid), then the register content, tag, and busy bit are copied to the reservation station (at sink or source fields). A new tag is updated to the destination register in FLR, and the busy bit is set. This new tag is the reservation station's assigned number. However, if the busy bit of the operand register in FLR is already set (indicating another instruction is currently generating the register value), then only the register tag and busy bit are copied to the reservation station.

Detailed Description Paragraph Right (58):
Tomasulo's algorithm lacks a mechanism to handle speculative execution. If instructions are allowed to be fetched, decoded, and executed speculatively, then the FLR only contains the architectural state. Only the most recent updates of registers are maintained, regardless of whether they are speculative updates or not. For speculative execution to work properly, there must be a mechanism to recover certain old values and restart at the in-order state prior to the speculative branching. To support speculative execution, the register file can be accompanied by a reorder buffer. The original Tomasulo's algorithm also lacks multi-instruction fetch and decode. A single-instruction decoder really underutilizes the potential machine parallelism. It is not difficult however, to expand Tomasulo's reservation stations to handle multi-instruction decode and become a superscalar distributed window (see Johnson, 1991).

Detailed Description Paragraph Right (67):
The IBM RS/6000 is a multi-chip superscalar processor with a RISC instruction set (derivation of the 801 instruction set), capable of executing up to four instructions per cycle. FIG. 9 shows the architecture of the IBM RS/6000. There are three functional units--the branch unit (BU), fixed-point unit (FXU), and floating-point unit (FPU)--that are capable of executing instructions in parallel. The BU can issue up to two instructions per cycle, a branch and a condition-register instruction. The FXU issues one instruction per cycle, which can be a fixed-point arithmetic, a fixed-point load/store, or a floating-point load/store. The FPU issues one floating-point arithmetic instruction per cycle including a multiply-add instruction. Each functional unit has instruction buffers (I-buffers) to shelve instructions. These I-buffers are organized as a FIFO instruction window with in-order issue. The BU's I-buffers are a central window that contain all fetched instructions (not decoded yet). The FXU's and FPU's I-buffers are distributed windows. Both receive the same fixed- and floating-point instructions (not decoded at this point).

Detailed Description Paragraph Right (69):
The BU receives four instructions per cycle from the I-cache arrays into the BU's 12-entry I-buffers and dispatch unit. The dispatch unit dispatches externally to the FXU and/or FPU any two-instruction combination of fixed- and floating-point instructions. If the remaining two instructions contain a branch and/or a condition-register instruction, they are dispatched and executed internally in the BU. When a conditional branch instruction is encountered, the BU fetches the

branch-not-taken path instructions (default branch prediction direction) and dispatches them to the FXU and FPU. These instructions are executed speculatively and can be canceled if the branch is mispredicted (by postponing retirement to register files and flushing the pipelines). The branch-taken path instructions are also fetched from the I-cache arrays and placed at the BU's I-buffers, but their dispatching (or flushing) is held off until the branch outcome is known. The worst-case penalty for a branch misprediction is three cycles. The penalty can be eliminated if there are enough independent instructions to separate the compare from the branch.

Detailed Description Paragraph Right (70):
The BU has four special purpose registers (see R. R. Oehler and R. D. Groves, "The IBM RISC/6000 Processor Architecture," IBM Journal of Research and Development, Vol. 34, No. 1, pp. 23-36, 1990); the machine-state register (to control system states), the link register (for subroutine return address), the count register (to control loop iteration), and the condition register (to support multiple condition codes for conditional branches). Zero-delay branch for loops with a known iteration count is achieved with the branch-and-count instruction that uses the count register. The condition register contains eight condition fields, two of which are reserved to contain the condition code results of arithmetic computations in the FXU and FPU. The remaining six can be explicitly set by other fixed- or floating-point compare instructions and special branch-unit instructions. The setting of each condition field is controlled by the record bit (Rc) of arithmetic instructions. There are advantages to having multiple, setable condition fields. First, the compiler can schedule a compare instruction early, as far away as possible from the conditional branch instruction. Second, several compare instructions can be scheduled first (their results written into separate condition fields), which are then followed by a single conditional branch instruction. This is useful to implement a guarded statement/code section with multiple guard conditions, eliminating the typical structure of a series of single compares followed by a single branch.

Detailed Description Paragraph Right (71):
The FXU contains I-buffers, an arithmetic logic unit (ALU), a general-purpose fixed-point register file, and a single-entry store buffer. The I-buffers receive both fixed- and floating-point instructions from the BU's dispatch unit, but issue only fixed-point instructions and floating-point load/store instructions to the ALU. Addresses of all loads/stores are computed in the FXU. The ALU includes a multiply/divide unit with 3- to 5-cycle multiply and 19- to 20-cycle divide latencies. The store buffer holds data and address of one fixed-point store instruction. The store buffer makes load bypassing possible. Address and data of floating-point store instructions are buffered in the FPU. The I-buffer is a strictly FIFO instruction window with in-order issue. Partial out-of-order issue is achieved among different functional units. Since there is only one ALU and no out-of-order issue in the FXU, the integer RF is not accompanied by a result buffer. Result values are written directly to the integer RF, except on speculative results which are held off in the pipeline until the branch condition is known. Further instruction issue/execution in the FXU must be stalled. This limits the speculative execution capability in the IBM RS/6000.

Detailed Description Paragraph Right (72):
The FPU contains I-buffers, a unified floating-point multiply-add-fused unit (MAF), a floating-point register file, a register-mapping table, and a store buffer. FPU's I-buffers receive the same instructions as FXU's I-buffers, but issue only floating-point arithmetic instructions to the MAF. The MAF can perform an indivisible multiply-accumulate operation (A.times.B)+C, which reduces the latency for chained multiply-add operations, rounding errors, chip busing, and the number of adders/normalizers. The latency of a floating-point multiply-add instruction (FMA) is two cycles (see R. K. Montoye, et al., "Design of the IBM RISC System/6000 Floating-Point Execution Unit," IBM Journal of Research and Development, Vol. 34, No. 1, pp. 59-70, 1990). The register-mapping table provides register renaming (8 renaming registers) to allow independent, out-of-order executions of floating-point load/store and arithmetic instructions. The store buffer contains five entries for addresses and four entries for data. A floating-point store instruction is issued at the FXU where the store address is calculated and placed at the store buffer. Once the FPU produces the store value, it is placed at the corresponding entry in the

store buffer, ready to be committed to the data cache (it is to be understood that memory or other storage units may be used instead of a data cache). By buffering stores, the FXU can continue issuing subsequent loads (load bypassing).

Detailed Description Paragraph Right (74):
FIG. 10 shows the cycle-by-cycle execution of the inner loop. The superscripts indicate the iteration numbers. During cycle 1, four instructions (I7.sup.0, I8.sup.0, I9.sup.0, I10.sup.0) starting from_LOOP label are fetched from the I-cache arrays and placed into BU's I-buffers. During cycle 2, the first load and multiply-add instructions (17.sup.0, I8.sup.0) are sent to the FXU and FPU, respectively. The next four instructions are fetched (I11.sup.0, I12.sup.0, I13.sup.0, I14.sup.0). During cycle 3, the FXU decodes the floating-point load (I7.sup.0) and discards the floating-point multiply-add (18.sup.0). The FPU pre-decodes both instructions for register renaming. The loop-closing BCT instruction (I15.sup.0) is fetched. During cycle 4, there is no valid instruction fetch because the branch target address is still being computed. The FXU computes the address of the first load (I7.sup.0), while decoding the second load (I9.sup.0). The FPU renames the floating-point registers of I7.sup.0 and I8.sup.0. The BU detects the BCT instruction and generates the branch target address. During cycle 5, instructions from the next iteration (I7.sup.1, I8.sup.1, I9.sup.1, I10.sup.1) are fetched. The D-cache is accessed for the first load (I7.sup.0). The FXU computes the address of the second load (I9.sup.0). The first FMA instruction (I8.sup.0) is decoded at the FPU. During cycle 6, the FPU executes the first FMA instruction while decoding the second FMA instruction (I10.sup.0). The D-cache is read for the second load (I9.sup.0). In summary, the first iteration outputs x(i)' and y(i)' are stored at cycle 10 and 11, respectively. The iteration period of this loop is 4 cycles. In FIG. 10, there is no branch penalty (zero-cycle branch) in FXU's and FPU's pipelines. The execute pipeline stages (FXE, FPE1, FPE2) are always full, primarily because the instruction fetch rate is twice the instruction issue rate at the arithmetic units. However, a true branch penalty should be seen at the fetch stage (IF), which in this case shows a one-cycle branch delay due to the branch address calculation.

Detailed Description Paragraph Right (75):
The IBM RS/6000 processor chipset consists of nine chips (including I-cache and D-cache arrays, bus controller, and I/O controller), which are implemented in a 1-.mu.m, three-metal CMOS process. The total number of transistors is 6.9 million. The benchmark performance figures on the top of the line system, the IBM RS/6000 POWERSTATION (Trademark of IBM Corporation) 580 (62.5 MHz), are SPECint92 61.7, SPECfp92 133.2 (see R. Myrvaagnes, "Beyond Workstations," Electronic Products, pp. 17-18, 1993), and 38.1 MFLOPS on (double precision, N=100) LINPACK (Trademark of Digital Equipment Corporation--see Digital Equipment Corporation, ALPHA AXP Workstation Family Performance Brief-Open VMS, 1992).

Detailed Description Paragraph Right (76):
In general, the IBM RS/6000 supports all six superscalar features (multi-instruction issue, decoupled dataflow scheduling, out-of-order execution, register renaming, speculative execution, and precise interrupts), some in a limited way. Although four instructions are fetched per cycle, only two (fixed- and floating-point) or three (including a branch) instructions are typically issued per cycle. Only single-level speculative execution is supported. Multiple unresolved conditional branches cause issue stalls because of the lack of a result buffer. Precise interrupts are not supported in the regular mode. They are only supported when the processor is put in the "synchronize" mode, which slows the processor significantly.

Detailed Description Paragraph Right (77):
The LIGHTNING SPARC microprocessor, from Metaflow Technologies, Inc., is the first implementation of the Metaflow architecture that executes the SPARC (v.8) RISC instruction set. The architecture is based on the DCAF (dataflow content-addressable FIFO), a DRIS implementation. Although the DRIS is conceptually a central window, it is implemented as three physical windows: (a) the central DCAF in DIU which shelves all instructions (complete DRIS), (b) the floating-point DCAF in FPU which shelves only floating-point instructions, and (c) the branch DCAF in IIU which shelves only conditional branch instructions. The central DCAF is the central window that is responsible for retiring operations; while others are only a subset of the central

DCAF. FIG. 11 shows the LIGHTNING SPARC module which consists of an external cache (up to 1 Mbyte) and four ASICs; the instruction issue unit (IIU), the dataflow integer unit (DIU), the floating-point unit (FPU), and the cache controller/MMU/bus interface (CMB). The external cache consists of the first-level cache for data and the second-level cache for instructions (the first-level is in the IIU chip).

Detailed Description Paragraph Right (79):
The IIU's branch unit executes a conditional branch instruction speculatively, and shelves it at the branch DCAF for a later misprediction check. The branch DCAF is a specialized DRIS implementation that shelves conditional branch instructions that were speculatively executed. The ID of the oldest, unresolved conditional branch instruction is sent to the retire logic in the central DCAF, to prevent retiring speculative instructions. During the writeback stage, the IDs and results of up to three condition code-setting instructions (two from the integer ALUs in DIU and one from the floating-point adder in FPU) are updated to the branch DCAF. During the execute stage, all branch entries with unlocked operands compare their condition code result with the prediction. If not the same, then the conditional branch was misspeculated. Branch repair is initiated on the oldest mispredicted branch. Its ID is broadcast to all DCAFs. All entries with younger IDs are flushed by moving the tail pointers accordingly.

Detailed Description Paragraph Right (80):
The DIU contains the central DCAF, two integer ALUs, one memory-address ALU, and retire logic. Up to three instructions can be allocated into the central DCAF, including floating-point instructions which are used for retirement purposes. The central DCAF invokes issue and writeback operations only on integer and memory-reference instructions. To allow proper retiring, each cycle the central DCAF is informed by the FPU on all IDs of successfully executed floating-point instructions. The retire logic can retire up to eight instructions per cycle in the central DCAF (see Popescu, et al.): three that update integer registers or condition codes, two that update floating-point registers, one store instruction, and any two instructions of other types (control transfers, processor state updates). The FPU contains the floating-point DCAF, a floating-point adder, and a floating-point multiplier. The floating-point DCAF invokes allocate, issue, and writeback only on floating-point instructions. To deallocate entries, the floating-point DCAF is informed by the DIU on the IDs of retired floating-point instructions.

Detailed Description Paragraph Right (84):
The SUPERSPARC processor from Texas Instruments, Inc. is the first commercial superscalar implementation of the SPARC version 8 architecture (Sum Microsystems Computer Corporation, The SUPERSPARC Microprocessor-Technical White Paper, 1992). A virtually identical version from Ross Technology, Inc. and Cypress Semiconductor Corporation is called the HYPERSPARC (Trademark of Ross Technology, Inc. and Cypress Semiconductor Corporation). FIG. 12 shows the SUPERSPARC architecture which primarily consists of three functional units: an instruction unit, integer unit, and floating-point unit. There are also a 20-Kbyte I-cache that fetches four instructions per cycle, and a 16-Kbyte D-cache that can handle one 64-bit load or store per cycle. These on-chip caches can interact with the MBus or a second-level cache controller that supports up to 1-Mbyte of external cache.

Detailed Description Paragraph Right (87):
The floating-point unit provides a 4-entry floating-point instruction queue, 5-port floating-point register file, floating-point adder, and floating-point multiplier. A floating-point instruction is issued from the bottom (oldest entry) of the instruction queue when the operands and resources are available. All floating-point instructions start in order and complete in order (see Sun Microsystems Computer Corporation, 1992). The floating-point adder performs addition, subtraction, format conversion, comparison, absolute value, and negation. The floating-point multiplier performs single- and double-precision multiplication, division, square root, and integer multiplication and division. Bypassing capabilities from the result buses and load bus to arithmetic units are provided. The latency of most floating-point operations is three cycles.

Detailed Description Paragraph Right (89):
The SUPERSPARC processor supports five superscalar features; multi-instruction

issue, decoupled dataf low scheduling, speculative execution, out-of-order execution, and precise interrupts. Speculative execution is handled by holding a speculative result at the end of the pipeline before being written to the register file (until the conditional branch is resolved). If mispredicted, all instructions and results currently in processing pipelines are flushed. The multiple-path fetching into the sequential and target queues helps reduce the branch misprediction penalty. The SUPERSPARC architecture is still somewhat limited in its superscalar capabilities: (1) The multi-instruction issue has a lot of restrictions/rules, and is limited to three instructions despite the four-instruction fetch. Output dependencies also stall instruction issue because register renaming is not supported.

Detailed Description Paragraph Right (91):
(3) Only limited out-of-order execution is supported; no load bypassing, and strictly in-order issue with the possibility of out-of-order completion of floating-point instructions from the floating-point queue with respect integer instructions.

Detailed Description Paragraph Right (92):
The DEC ALPHA 21064 processor is the first implementation of Digital Equipment Corporation's 64-bit ALPHA AXP architecture (see E. McLellan (Digital Equipment Corporation), "The APLHA AXP Architecture and 21064 Processor," IEEE Micro, pp. 36-47, 1993). It is currently the fastest single-chip microprocessor in the industry. The architecture is a combination of superpipelined and superscalar architectures. The integer and floating-point pipelines are seven- and ten-stages deep, respectively. Since DEC has an existing, large customer base of software, it offers compatibility with VAX and MIPS codes through binary translation. Executable program codes are converted to AXP code without recompilation (with some performance degradation). FIG. 13 shows the DEC ALPHA 21064 architecture, which has four functional units: an instruction unit (IBox), an integer unit (EBox), a floating-point unit (FBox), and an address unit (ABox). There are also 32 entry by 64-bit integer and floating-point register files (RFs), 8-Kbyte D-cache, and 8-Kbyte I-cache with a 2 K by 1-bit branch history table. The branch history table is provided for dynamic prediction and achieves 80% accuracy on most programs. Static prediction is also supported based on the sign of the branch address displacement field as the default; backward branches are predicted taken and forward branches are predicted not-taken.

Detailed Description Paragraph Right (94):
The EBox contains dedicated integer multiplier, adder, shifter, and logic units. The multiplier unit is not pipelined to save silicon area. The adder and logic units have single-cycle latency with bypass paths for register write data. The shifter takes two cycles to produce results, but is fully pipelined (one-cycle throughput). The FBox contains dedicated floating-point multiplier/adder and divider units. It supports both VAX- and IEEE-standard data types and rounding modes. The divider unit generates one bit of quotient per cycle. All other floating-point operate instructions have six-cycle latency and one-cycle throughput.

Detailed Description Paragraph Right (96):
FIG. 14 shows the pipeline stages of the DEC ALPHA 21064 processor for integer and floating-point instructions. Up to two instructions can be processed in each stage. The first three stages (IF, SW, I0) can be stalled, while stages beyond I0 advance every cycle (see D. W. Dobberpuhl, et al., "A 200-MHz 64-Bit Dual-Issue CMOS Microprocessor," Digital Technical Journal, Vol. 4, No. 4, Special Issue, pp. 35-50, 1992). In stage IF, a pair of instructions is fetched from the on-chip I-cache. In stage SW, a swap or serialization operation is performed based on the dual-issue rules. If there is a conditional branch instruction, the branch direction is predicted statically or dynamically (using the branch history table). In stage I0, instruction(s) are decoded and checked for dependencies between the two fetched instructions (if any). In stage I1, instruction(s) are issued to the appropriate functional unit, provided there is no register conflict. The source operands are read from the integer and/or floating-point RFs and sent to the EBox, IBox, ABox, and FBox. In stage 4, instruction executions start (stage A1 for integer instructions, stage F1 for floating-point instructions).

Detailed Description Paragraph Right (97):
FIG. 14(a) shows that all integer arithmetic and logic instructions (EBox), except shift instructions, have one-cycle latency, through bypass paths. Shift instructions have two-cycle latency. All results in EBox are actually written back to the integer RF in stage 6. Without the bypass path, the latency would be three cycles. But with the bypass path, the latency is reduced to one or two cycles. This improves the probability that back-to-back dependent instructions execute at full pipeline speed. The DECchip 21064 dedicates 45 different bypass paths. Conditional branch instructions (IBox) are resolved in stage 4. If a branch misprediction is detected, a branch repair is initiated. Instructions subsequent to the branch (in the wrong path) and their intermediate results are flushed from all pipeline stages. The alternate branch target address is computed as the new PC. The first instruction pair of the correct branch path is fetched at stage 6. This branch misprediction causes a four-cycle delay. Primary, on-chip D-cache accesses of load and store instructions (ABox) complete in stage 6. So, the latency of loads and stores is three cycles. FIG. 14(b) shows that results of floating-point operations (from the multiplier/adder unit) are written back to the floating-point RF in stage 9, thus giving a 6-cycle latency. The ALPHA AXP architecture has several notable characteristics:

Detailed Description Paragraph Right (98):
The DEC ALPHA 21064 single-chip processor is implemented using a 0.75 .mu.m, three-metal CMOS process, with operating speeds up to 200 MHz. The extremely high clock frequency presents a difficult clocking situation. To avoid race conditions for latched data, the clock edge rate must be extremely fast (0.5 ns) and only very little clock skew can be tolerated. DEC's solution is to implement a very large, on-chip clock driver with a final stage containing 156 to 172-mil-wide PMOS and 63 to 78-mil-wide NMOS devices (see McLellan). The clock driver occupies about 5% of the total chip area and draws a peak switching current of 43 A. A 0.13-.mu.F on-chip decoupling capacitance must be added to overcome the supply voltage problem. The chip's power dissipation is 30 W at 200 MHz with a 3.3-V supply. Sophisticated packaging is used to cool the chip. These hardware cost and implementation problems are compensated by top performance. The benchmark performance figures on the top-of-the-line system, the DEC 10000/610 (200 MHz), are: SPECint92 116.5, SPECfp92 193.6, and 40.5 MFLOPS on 100.times.100 Linpack (double precision).

Detailed Description Paragraph Right (104):
The HP PA-7100 processor is the seventh implementation of Hewlett-Packard's PA-RISC (precision architecture, reduced instruction set computer--Trademark of Hewlett-Packard, Inc.) architecture (T. Asprey, et al. (Hewlett-Packard), "Performance Features of the PA-7100 Microprocessor," IEEE Micro, pp. 22-35, 1993). It is the first superscalar PA-RISC design, which issues up to two instructions per cycle. Its design also has a VLIW flavor. There are two notable design approaches in the PA-RISC architecture; (a) the use of off-chip, rather than on-chip, primary caches (I-cache and D-cache), and (b) the reduction of instruction count in programs (pathlength reduction--see R. Lee, et al., "Pathlength Reduction Features in the PA-RISC Architecture," Proceedings of the 37th COMPCON, pp. 129-135, 1992) by adding VLIW-like and SIMD-like instructions. The motivation to use off-chip caches is the fact that on-chip caches are usually not large enough to achieve balanced performance across a wide range of applications. Typically, on-chip I-caches range from 8 to 20 Kbytes, and on-chip D-caches range from 8 to 16 Kbytes. The PA-7100 processor can have up to 1 Mbyte I-cache and 2 Mbyte D-cache. Unlike most processors with small on-chip caches, a secondary cache becomes unnecessary. Another advantage is the flexibility of cache size and speed to configure different systems, from low-end to high-end systems.

Detailed Description Paragraph Right (105):
The objective of pathlength reduction is to resolve the key disadvantage of RISC architectures, the code/pathlength expansion. There are two instruction types added to the RISC instruction set. First, two or three operations that frequently occur together are combined into a fixed-length, 32-bit instruction. This results in multi-operation, VLIW-like instructions (except they are contained within a short 32-bit instruction), such as Shift&Add (perform integer multiplications with a small constant), Multiply&Add (floating-point), Compare&Branch, Add&Branch, Branch_on_Bit, etc (see Lee, et al.). Other streamlined RISC architectures such as MIPS require

multiple instructions to perform these tasks. Second, SIMD-like instructions are added to operate, in parallel, on multiple data units smaller than a 32-bit word. These instructions are particularly useful in parallel character and decimal operations. For example, in the C language, character manipulations frequently involve finding the null byte (zero) that marks the end of a variable-length string of characters. PA-RISC's Unit_Exclusive_Or instruction speeds this process by testing a "no byte zero" in a word of four bytes in a single cycle (see Lee, et al.). The addition of the two instruction types is accommodated in the hardware without impacting the cycle time or the CPI. This gives the PA-RISC architecture some of the advantages of a very simple VLIW architecture (with short 32-bit instructions), without losing the advantages of a RISC architecture.

Detailed Description Paragraph Right (106):
FIG. 15 shows the PA-7100 architecture. The processor chip consists of six major blocks; the integer unit, floating-point unit, cache control/interface, unified TLB, control unit, and system bus interface. The control unit is responsible for fetching, decoding, and issuing of instructions. Two instructions are fetched from the off-chip I-cache per cycle, and buffered in a small prefetch buffer (central window). The control unit can issue up to two instructions per cycle, one to the integer unit and one to the floating-point unit. There are no alignment or order constraints on the pair of instructions (see E. DeLano, et al., "A High Speed Superscalar PA-RISC Processor," Proceedings of the 37th COMPCON, pp. 116-121, 1992). However, no two integer or floating-point instructions can be issued simultaneously. If a conditional branch instruction is encountered, a simple static branch prediction scheme is used to minimize branch penalty. All forward conditional branches are untaken and backward conditional branches are taken.

Detailed Description Paragraph Right (107):
The integer unit contains an ALU, shift-merge unit (SMU), dedicated branch adder, and a 32.times.32-bit, general-purpose, integer register file. Besides integer arithmetic instructions, the integer unit also executes branch instructions, loads and stores of integer and floating-point registers, and all VLIW-like and SIMD-like instructions, except the floating-point Multiply&Add and Multiply&Subtract. The VLIW-like instructions improve the utilization of the three hardware units. For example, the Add&Branch uses the ALU and the branch adder simultaneously, while the Branch_on_Bit uses the SMU and the branch adder (see Lee, et al.). The register bypass paths produce a one-cycle latency for integer arithmetic instructions.

Detailed Description Paragraph Right (108):
The floating-point unit contains a floating-point ALU (FALU), multiplier (FMUL), divide/square root unit (FDIV/SQRT), and a 32.times.64-bit floating-point register file. Although there are 32 physical registers, the first four registers (0-3) are dedicated for status register and exception registers. The remaining 28 registers (4-31) are used as register operands for arithmetic operations. Each register can be access as a 64-bit double word or as two 32-bit single words. The FALU performs single- and double-precision add/subtract, compare/complement, and format conversion instructions. The FMUL performs single- and double-precision multiplications, and also 32-bit unsigned integer multiplications (64-bit result). The multiplier array is based on a radix-4 Booth encoding algorithm. The register bypass paths produce a two-cycle latency for all floating-point instructions performed in the FALU and FMUL. The FDIV/SQRT performs floating-point divide and square-root operations based on a modified radix-4 SRT (Sweeney, Robertson, and Tocher--see Asprey, et al.) algorithm. The main modification is running the radix-4 division hardware at twice the processor clock frequency to effectively achieve a radix-16 performance. Four quotient bits are computed each clock cycle, giving a latency of 8 and 15 cycles for single- and double-precision divide/square root operations. The floating-point register file has five read ports and three write ports to allow concurrent execution of a floating-point multiply, a floating-point add, and a floating-point load or store. This occurs when a Multiply&Add or a Multiply&Subtract instruction is issued concurrently with a floating-point load/store instruction (categorized as an integer instruction).

Detailed Description Paragraph Right (109):
The instruction execution pipeline for various types of instructions is shown in FIG. 16. The pipeline frequency is determined by the read cycle time of the off-chip

cache RAMs (see Asprey, et al.). Each pipeline stage is divided into two equal phases (2-phase clocking scheme). The first three phases are dedicated for instruction fetching from the off-chip I-cache. Instruction decode and issue can be done in a mere single phase because a pre-decoded bit is dedicated in the instruction-field format to steer instructions to the integer and floating-point units. The phases for instruction execution depend on the instruction type, as depicted in FIG. 16. For a conditional branch instruction, instructions along the predicted path are fetched (static branch prediction) while the branch condition is evaluated. In the meantime the alternate link address (Laddr) is also calculated. If at the end of the execute stage the branch is found to be mispredicted, the previous speculative instruction fetch is flushed and new instructions along the correct path are fetched. If the delay is viewed from the I-fetch to the Target I-fetch, the minimum branch delay of correctly and incorrectly predicted branches is one cycle and two cycles, respectively. The PA-7100 processor has extensive register bypass capability to minimize pipeline interlock penalties. As illustrated in FIG. 16, the penalty for integer ALU pipeline interlock is zero cycles. The penalty for load use, floating-point ALU, or floating-point multiply pipeline interlocks is one cycle.

Detailed Description Paragraph Right (110):
The HP PA-7100 processor is implemented using a 0.8 .mu.m, three-metal CMOS process. It operates at 100 MHz and integrates about 850,000 transistors. The use of off-chip caches results in a large pin-count package, 504-pin PGA. The reported benchmark performance figures on the top-of-the-line system, the HP9000/735 (99 MHz), are: SPECint92 80.0, SPECfp92 150.6, and 40.8 MFLOPS on 100.times.100 Linpack (double precision).

Detailed Description Paragraph Right (111):
The HP PA-7100 processor supports four superscalar features; multi-instruction issue, decoupled dataflow scheduling, out-of-order execution, and precise interrupts. However, it does not support register renaming and speculative execution. Note that static branch prediction is only used for speculative prefetch, not speculative execution. Instructions following an unresolved conditional branch are stalled and not executed. The HP PA-7100 designers rely on aggressive VLIW-like software scheduling and chose not to push the superscalar hardware too aggressively:

Detailed Description Paragraph Right (112):
(1) The multi-instruction issue is limited to dual issue of integer and floating-point instruction pairs. No two integer or floating-point instructions can be issued simultaneously. To increase machine parallelism, the VLIW-like and SIMD-like instructions are included, which increases the complexity of the compiler.

Detailed Description Paragraph Right (115):
The Intel PENTIUM microprocessor is the first superscalar implementation that runs the widely-used x86 CISC instruction set. The x86 instructions use only two operands and permit combinations of register and memory operands. Thus, unlike all other commercial superscalar processors, the PENTIUM processor is not a typical register-to-register, three-address machine. Despite the complexity of CISC instructions, many of which require microcode sequencing, the PENTIUM processor manages to differentiate the "simple" (RISC-like) instructions and executes them in superscalar mode (dual-instruction issue). However, complex instructions and almost all floating-point instructions must still run in scalar mode (single-instruction issue). The superscalar execution and architectural improvements in branch prediction, cache organization, and a fully-pipelined floating-point unit result in a substantial performance improvement over its predecessor, the i486 processor. When compared with an i486 processor with identical clock frequency, the PENTIUM processor is faster by factors of roughly two and five in integer and floating-point performance, respectively (D. Alpert and D. Avnon--Intel Corporation, "Architecture of the PENTIUM Microprocessor," IEEE Micro, pp. 11-21, 1993).

Detailed Description Paragraph Right (116):
FIG. 17 shows the PENTIUM architecture. The core execution units are two integer ALUs and a floating-point unit with dedicated adder, multiplier, and divider. The prefetch buffers fetch a cache line (256 bits) from the I-cache and performs

instruction aligning. Because x86 instructions are of variable length, the prefetch buffers hold two cache lines; the line containing the instruction being decoded and the next consecutive line (see Alpert and Avnon). An instruction is decoded and issued to the appropriate functional unit (integer or floating-point) based on the instruction type. Two instructions can be decoded and issued simultaneously if they are "simple" instructions (superscalar execution). Because x86 instructions typically generate more data memory references than RISC instructions, the D-cache supports dual accesses to provide additional bandwidth and simplify compiler instruction scheduling algorithms (see Alpert and Avnon).

Detailed Description Paragraph Right (119):
The floating-point unit consists of six functional blocks: the floating-point interface, register file, and control (FIRC), the floating-point exponent (FEXP), the floating-point multiplier (FMUL), the floating-point adder (FADD), the floating-point divider (FDIV), and the floating-point rounder (FRND). The FIRC contains a floating-point register file, interface logic, and centralized control logic. The x86 floating-point instructions treat the register file as a stack of eight registers, with the top of the stack (TOS) acting as the accumulator. They typically use one source operand in memory and the TOS register as the other source operand as well as the destination register. In the case of 64-bit memory operands, both ports of the D-cache are used. To swap the content of the TOS register with another register, the FXCH instruction (non-arithmetic floating-point instruction) is used. The FIRC also issues floating-point arithmetic instructions to the appropriate arithmetic blocks. Non arithmetic floating-point instructions are executed within the FIRC itself. Floating-point instructions cannot be paired with any other integer or floating-point instructions, except FXCH instructions.

Detailed Description Paragraph Right (120):
The FEXP calculates the exponent and sign results of all floating-point arithmetic instructions. The FADD executes floating-point add, subtract, compare, BCD (binary coded decimal), and format conversion instructions. The FMUL executes single-, double-, extended-precision (64-bit mantissa) floating-point multiplication and integer multiplication instructions. The FDIV executes floating-point divide, remainder, and square-root instructions. And, the FRND performs the rounding operation of results from FADD and FDIV. The floating-point unit also supports eight transcendental instructions such as sine (FSIN), cosine (FCOS), tangent (FPTAN), etc. through microcode sequences. These instructions primarily involve the FADD arithmetic block and sometimes other arithmetic blocks.

Detailed Description Paragraph Right (121):
The PENTIUM processor is implemented using a 0.8 .mu.m BiCMOS process. It integrates 3.1 million transistors and currently runs at 66 MHz. The integer pipeline consists of five stages: prefetch (PF), first decode (D1), second decode (D2), execute (E), and writeback (WB). The floating-point pipeline consists of eight stages, where the first three stages (FP, D1, and D2) are processed with the resources in the integer pipeline. The other floating-point stages are: operand fetch (E), first execute (X1), second execute (X2), write float (WF), and error reporting (ER). The reported benchmark performance figures of the PENTIUM processor are: SPECint92 64.5 and SPECfp92 56.9.

Detailed Description Paragraph Right (124):
Output dependencies (artificial dependencies) stall instruction issue because register renaming is not supported. Floating-point instructions also cannot be paired with any other instructions, except occasionally with FXCH instructions (but FXCH may be considered as a useless or unnecessary instruction in true register-to-register architectures). The multiple floating-point arithmetic blocks (FADD, FMUL, FDIV) are underutilized by the limitation of one floating-point instruction per cycle.

Detailed Description Paragraph Right (127):
(4) The out-of-order execution is limited to in-order issue with the possibility of out-of-order completion between instructions in the integer and floating-point pipelines. Load bypassing is also not supported.

Detailed Description Paragraph Right (129):

Table 4 is a summary of upcoming commercial superscalar microprocessors in 1995/1996. DEC continues to lead the pack with its new ALPHA 21164 design. The major architectural improvements from its ALPHA 21064 predecessor are quad-issue, additional integer and floating-point execution units (total 2 each), and the inclusion of a secondary cache on chip (see L. Gwennap, "Digital Leads the Pack with 21164," Microprocessor Report, Vol. 8, No. 12, pp. 1 and 6-10, Sep. 12, 1994). The last feature is the first in the history of microprocessors and makes the ALPHA 21164 the densest with 9.3 million transistors. Sun Microsystems' ULTRASPARC (Trademark of Sun Microsystems Computer Corporation) incorporates nine independent execution units, including dedicated graphics add and multiply units. The ULTRASPARC is the first implementation of the new 64-bit SPARC version 9 instruction-set architecture, which supports conditional move instructions. It also supports MPEG-2 graphics instructions in hardware to boost multimedia application performance. The IBM POWERPC 620 is the latest and currently fastest among other PowerPC models (601, 603, 604, 615). It uses reservation stations to shelve instructions at six execution units (see L. Gwennap, "620 Fills Out POWERPC Product Line," Microprocessor Report, Vol. 8, No. 14, pp. 12-17, Oct. 24, 1994). IBM put two entries for each execution unit with the exception at load/store unit (3 entries) and branch unit (4 entries). The MIPS Technologies' R10000, also known as T5, uses "decoupled architecture" (another term for decoupled dataf low scheduling) with a set of three central windows (16-entry queues) for memory, integer, and floating-point instructions (see L. Gwennap, MIPS R10000 Uses Decoupled Architecture," Microprocessor Report, Vol. 8, No. 14, pp. 17-22, Oct. 24, 1994). The R10000 uses a register-mapping table (also called rename buffer) to support register renaming. Both integer and floating-point units have 64, 64-bit physical registers that are mapped to 32 logical registers. To handle multi-level speculative execution (up to 4 conditional branches), the R10000 saves the mapping table in shadow registers when encountering a conditional branch. The HP PA-8000 is the first 64-bit PA-RISC architecture implementation (see L. Gwennap, "PA-8000 Combines Complexity and Speed," Microprocessor Report, Vol. 8, No. 15, pp. 1 and 6-9, Nov. 14, 1994). Like its predecessors (PA-7100, PA-7100), the PA-8000 will not have on-chip cache. PA-RISC is the only advanced, general-purpose microprocessor architecture that uses off-chip L1 cache. The AMD K5 is currently the fastest x86 processor, claimed to be at least 30% faster than the Intel PENTIUM at the same clock rate (on integer code--see M. Slater, "AMD's K5 Designed to Outrun PENTIUM," Microprocessor Report, Vol. 8, No. 14, pp. 1 and 6-11, 1994). Despite the CISC x86 instruction set, the architecture internally runs RISC instructions, called ROPs (RISC operations). To achieve this, x86 instructions are predecoded as they are fetched from memory to the I-cache. The predecoder adds five bits to each byte, causing an increase of about 50% at the I-cache array. The K5 applies all the superscalar techniques that Johnson believed to be the best, the reservation station for instruction shelving and the reorder buffer for result shelving (see Johnson, 1991).

Detailed Description Paragraph Right (131):
Assume the processor has eight execution units: a branch unit, two fixed-point ALUs, a floating-point ALU, a floating-point multiply/divide/square-root unit, two load/store units, and a fixed/floating-point move unit. The processor begins by fetching at least one instruction or multiple instructions (N.sub.d instructions in this case, which is the decoder size) from the I-cache. It is to be understood that one or more memories or other storage units may be employed instead of the I-cache for performing the same function as the I-cache. These instructions are decoded in parallel and dispatched to their respective execution unit's reservation station. For each decoded instruction, an entry is allocated at the RB to shelve its result. To read its operand(s), each operand's register number is presented to the RB and register file, in which three situations can occur. First, if there is a matched entry in the RB and the register operand value has been calculated, then the operand value is routed/copied to the instruction's reservation station. Second, if there is a match entry in the RB but the value has not finished calculation, then the operand tag is copied instead to the reservation station. Third, if there is no match entry in the RB then the value from RF is the most recent one and copied to the reservation station.

Detailed Description Paragraph Right (133):
During the writeback stage, the execution result is written to the RB (not RF) and also forwarded to any reservation station waiting for this result value. In every

cycle, each valid reservation station with unavailable operand(s) compares its operand tag(s) with all result tags to determine when to grab certain result value(s). Note that if each execution unit's output port does not have a dedicated result bus (N.sub.res <fixed-point output ports, or N.sub.resfp <floating-point output ports), then arbitration logic must be provided to resolve who can use the shared result buses at a given time.

Detailed Description Paragraph Right (149):
Ideally we want to use the same result register tag, which is used during the result write operation, as the unique associative key for read operation. This tag is written to the RB entry during allocation. Smith and Plezkun use the RB identifier or array index as the result tag. But this tag is not unique with the presence of a second RB (e.g., for a floating-point register file). Moreover, the tag will keep changing as the FIFO queue is advanced during multi-entry retire. Tracking many different register tags plus conditional branch tags can be a nightmare. Weiss and Smith suggested a more flexible way of assigning unique result tags, which was originally proposed to be used in reservation stations (see above section entitled Reservation Stations). When an instruction is decoded, a new tag or identifier (inst_ID) is assigned from a "tag pool" that consists of some finite set of tags. Each destination register is then tagged with the inst_ID of the producer instruction. When the instruction completes, the inst_ID is returned to the pool for reuse. This tag pool, called the Instruction ID Unit (IIU), can be implemented as a circular instruction array (IA). The inst_ID is composed of (color_bit, IA_index--these are discussed in more detail below), the current "color" bit appended with its IA index (entry address), the same scheme used in the DRIS technique (see above section entitled DRIS). The color bit is used to distinguish the age or order of instructions when the valid entry area wraps around.

Detailed Description Paragraph Right (150):
Now the question is, how do we get the source operand tag to be used as the unique associative key when reading the RB, without reading the RB first? Remember that each decoded instruction with register operands needs operand tags (or operand values in the case of a reservation station technique) before being dispatched to an instruction window. To accommodate these operand tags at the decode stage, a Register Tag Unit (RTU) is added in the fetch and decode unit. Each decoded instruction presents its source register numbers to the RTU to get the corresponding register tags. The RTU can be viewed as a small register file that maintains the most recent tag (not value) of every register. The most recent tag of a register is the inst_ID of a producer instruction that updates the register last. When an instruction is assigned an inst_ID by the IIU, the destination register entry in the RTU is written with the inst_ID.

Detailed Description Paragraph Right (170):
The reservation station (RS) technique, currently considered the best technique, was originally introduced by Tomasulo in 1967 in the floating-point section of the IBM 360/91 (see Tomasulo). The main objective was to permit simultaneous execution of independent instructions while preserving the precedence (data dependency) constraints in the instruction stream. Tomasulo's RS technique was essentially ahead of its time. It actually accomplishes several superscalar objectives; multi-instruction issue, decoupled dataflow scheduling, out-of-order execution, and register renaming (eliminating anti and output dependencies). Anti dependencies (write-after-read hazards) are avoided by tagging registers, copying operands to reservation stations, and forwarding results directly to reservation stations. Output dependencies (write-after-write hazards) are avoided by comparing tags at the FLR (floating-point register unit in the IBM 360/91) on every register write, to ensure that only the most recent instruction changes the register. However, Tomasulo's algorithm lacks a mechanism to handle speculative execution. Only the most recent updates of registers are maintained, regardless of whether they are speculative updates or not. To support multi-level speculative execution, the register file can be accompanied by a reorder buffer (RB) as seen in the AMD superscalar 29 K (see Case) and K5 (see Slater), or multiple copies of register-mapping tables (RMT) as seen in IBM POWERPCs and MIPS R10000 (see Gwennap, Oct. 24, 1994).

Detailed Description Paragraph Right (171):

As we can see above, there is an overlap of task since both the RS and RB/RMT support register renaming. And we know that RS's efforts to support register renaming by operand value copying and result value forwarding actually penalize the RS implementation, due to excessive shared-global wires, comparators, and multiplexers. Therefore, if we already have RB or RMT or MRB result shelving, it seems logical to eliminate the expensive operand value copying and result value forwarding concept. The key concept is that no register values are stored in the DIQ, only their register tags. This eliminates large amount of global buses and wide-bus multiplexers or tristate buffers for data routing. Operand values are read directly during the issue/execute stage from the reorder buffer or register file, when they are available. Unlike the RS technique which reads operand values during decode stage, the DIQ technique does not have the result forwarding hazard problem. Therefore, we save a substantial number of comparators and wide-bus multiplexers (no operand value bypassing).

Detailed Description Paragraph Right (175):
FIG. 27 shows in-order issue DIQ 300 structure with N.sub.diq entry cells 395 and N.sub.d allocate ports 310, implemented as a circularly addressed register array. It has multiple allocate ports 310 and a single issue port 340. The DIQ 300 entry fields 385 vary from one execution unit to another with the first two fields, 386 and 387 (inst_ID and opcode) always present. The example shown is of a floating-point ALU which consists of: instruction tag 386 (N.sub.tag bits), opcode 387 (N.sub.opc bits), source 1 register number 388 (log.sub.2 N.sub.frf bits, N.sub.frf is floating-point RF size), source 1 register tag 389 (N.sub.tag bits), source 2 register number 390 (log.sub.2 N.sub.frf bits), and source 2 register tag 3915(N.sub.tag bits). Note that in contrast to MRB 100 which only retires field values in field 191 of appropriate cells 195 in the retire operation, DIQ 300 issues all field values of fields 386, 387, 388, 389, 390, and 391 of a single cell 395 in order during the issue operation to instruction issue register 516 (see FIG. 33). Each DIQ entry cell 395 consists of D flip-flops (DFFs) or other storage units 305 (FIG. 28) to hold these DIQ entry fields and logic for the allocate operation as determined by allocate logic in DIQ cell 395.

Detailed Description Paragraph Right (178):
FIG. 30 shows an enhanced DIQ 400 structure of the same floating-point ALU's DIQ 300 example in FIG. 27 to allow full out-of-order issue, although a DIQ for any functional unit could be used. Note, as shown in FIG. 30 (and FIG. 27), some of fields 485 (385 in FIG. 27) vary with the type of functional unit. The enhancements in the DIQ 400 comprise the additions of: (1) "issued" 486, "RS1_rdy" 491, and "RS2_rdy" 494 flags or fields in each entry (fields 487, 488, 489, 490, 492, and 493 are identical to fields 386, 387, 388, 389, 390, and 391, respectively, of FIG. 27), (2) comparators (not shown) to match a result tag with all operand tags (RS1_tag and RS2_tag) to update their ready flags (RS1_rdy and RS2_rdy), and (3) an issue logic 475 circuitry to determine which instruction entry should be issued next.

Detailed Description Paragraph Right (179):
In the out-of-order issue DIQ 400 structure, an entry is still allocated from the tail side. Multiple tail pointers 460 are provided to handle multiple entry allocations in entry cells 495 per cycle. A newly allocated entry has its RS1_rdy 491 and RS2_rdy 493 fields initially set based on the operand value availability at decode time. These flags are updated during the writeback stage, by forwarding result tags 420 (NOT the result values as in the RS technique) to the appropriate functional units (for example, tags of floating-point results go only to selected functional units that use floating-point operands). These result tags 420 are compared to each entry's operand tags. A match will set the corresponding ready flag (RS1_rdy 491 or RS2_rdy 493) to TRUE.

Detailed Description Paragraph Right (181):
FIG. 31 shows superscalar processor 500 that utilizes DIQ 300 or 400 instruction shelving and MRB 100R or 100F (both have identical structure to MRB 100, but MRB 100R buffers fixed-point register values, while MRB 100F buffers floating-point register values) result shelving techniques. Processor 500 has execution units 560, 570, 580, 590, 595, 596, 505, and 506 analogous to the execution units shown in FIG. 18. Superscalar processor 500 as shown in FIG. 31 clearly reveals a significant reduction in global buses and multiplexers compared to the processor of FIG. 18. By

eliminating operand value copying and result value forwarding, shared-global operand buses and both shared-global result buses and wide-bus multiplexers are avoided, replaced by private-local (module-to-module) read buses and write buses 525, respectively. The only shared-global buses left are the required instruction dispatch buses 550 to deliver decoded instructions to every execution unit's DIQ 300 or 400. In the case of out-of-order issue DIQs 400 a small number of global wires to carry result tags are added (not shown in FIG. 31).

Detailed Description Paragraph Right (182):
In processor 500, separate register units 530 and 540 are provided for fixed-point and floating-point register results. (It is also possible to combine both types of register results in a single register unit.) With two register units, the processor 500 core area is basically segmented by Fixed-Point Register Unit (FXRU) 530 to hold general-purpose "R" registers, and the Floating-Point Register Unit (FPRU) 540 to hold floating-point "F" registers. Special purpose registers for condition codes can use any of the "R" registers, following the "no-single copy of any resource" philosophy of the DEC ALPHA architecture. A single-copy of any resource can become a point of resource contention. The "R" and "F" registers (which may be contained in either register units 530 and 540 in 515R and 515F or in MRB 100R and MRB 100F) are also used to hold fixed- and floating-point exception conditions and status/control information, respectively.

Detailed Description Paragraph Right (183):
Each of register units 530 and 540 contain register files 515R and 515F accompanied by MRBs 100R and 100F, respectively, to support register renaming, out-of-order execution, multi-level speculative execution, and precise interrupts. FIG. 32 shows the organization of a FXRU 530 (FPRU 540 is similar). RF 515R contains the in-order state, while MRB 100R contains the look-ahead state. An MRB 100R entry is retired to RF 515R only if it is safe. To read an operand, the register tag (reg_tag, a unique associative search key) is presented to MRB 100R and the register number (reg_num) is presented to RF 515R, thus performing the read on both RF 515R and MRB 100R. If a match entry is found in MRB 100R (read_found=1), then the register content in RF 515R is considered old and ignored. However, if the register is not found in MRB 100R then the RF 515R's read gives the correct register value. Note that finding a matched entry in MRB 100R does not guarantee that the register value has been found. The register value may still be computed in one of the execution units (560, 570, 580, 590, 595, 596, or 506) and has not been written to MRB 100R. MRB 100R includes allocate ports 110R, write ports 120R, read ports 130R, and retire ports 140R similar to MRB 100.

Detailed Description Paragraph Right (184):
Fixed-point arithmetic instructions are executed in the Fixed-Point Units (FXU 0 and FXU 1) 570 and 580. Floating-point arithmetic instructions are executed in the Floating-Point Arithmetic Logic Unit (FALU) 506 and Floating-Point Multiply/Divide/Sguare-Root Unit (FMDS) 505. Note that FALU 506 also performs floating-point compare/set instructions and writes its condition code directly to FXRU 530. Conditional/Immediate Move Unit (CIMU) 596 performs register move instructions between FXRU 530 and FPRU 540, as well as within FXRU 530/FPRU 540 itself. CIMU 596 can also be dedicated to handle conditional move instructions as seen in the SPARC-64 (version 9) instruction set. Load & Store Units (LSU 0 and LSU 1) 590 and 595 perform all load and store operations and include store queues (SQ) 591 and 594, respectively, to queue the store instructions until they can be committed/executed safely, with load bypassing and two simultaneous data accesses to D-cache 511 allowed. It is to be understood that one or more memories or other storage units may be employed instead of a cache for D-cache 511 for performing an equivalent function as D-cache 511. Branch instructions and PC address calculations are executed in the Instruction Address Unit (IAU) 560. A BTB (branch target buffer), which is a combination of a branch-target address cache (or other memory or storage unit) and branch history table, is provided in IAU 560 to help eliminate some branch delays and predict branch direction dynamically. During processor 500 implementation, it is best to physically layout circuit modules/blocks such that execution units 560, 570, 580, 590, 595, 596, 505, and 506 surround their corresponding register unit 530 or 540. Execution units that access both register units 530 and 540, such as LSUs 590 and 595 and CIMU 596, can be placed between the two. In this way, local bus 525 wiring is more direct and shorter.

Detailed Description Paragraph Right (186):
During normal operations, arithmetic and load/store instructions proceed through five processing steps/stages; fetch, decode, issue/execute, writeback, and retire. (Note that a stage does not necessarily represent a single hardware pipeline stage that can be performed in one clock cycle.) At the fetch stage, multiple instructions (N.sub.d) are fetched through fetch buffer 521 simultaneously from I-cache arrays (or other memory or storage unit) 510 (see FIG. 34). With instruction aligning done in I-cache arrays (or other memory or storage unit) 510 (as in the IBM RS/6000 processor--see Grohoski), N.sub.d instructions can be fetched each cycle, without wasted slots, as long as they reside in the same cache (or other memory or storage unit) 510 line. At the decode stage, multiple fetched instructions are decoded by instruction decoder 524 simultaneously. Each valid instruction is assigned a tag by IIU (Instruction ID Unit) 522, which is also used to tag the destination register of the instructions. An entry is allocated at the "R" (100R) or "F" (100F) MRB for each new register assignment. Register tags of an instruction's operands are acquired from the RTU (Register Tag Unit) 523. Finally, at least one valid decoded instruction or all (or multiple) valid decoded instructions are dispatched to the appropriate execution unit's (560, 570, 580, 590, 595, 596, 505, or 506) DIQ 300 or 400 Decoded instructions are shelved by DIQs 300 or 400 to allow more time to resolve data dependencies. Each DIQ 300 or 400 includes dependency check logic that automatically issues an instruction at the bottom of DIQ 300 or 400 as soon as its operands become available, independent of other DIQs 300 or 400. Unlike typical von Neumann processors, no centralized control unit is required to explicitly and rigidly sequence every instruction, deciding when it can execute. This is the essence of dynamic, dataflow scheduling. At burst situations, all execution units 560, 570, 580, 590, 595, 596, 505, and 506 simultaneously issue an instruction, achieving maximum machine parallelism. Results are not directly written back to their register file 515R or 515F, but shelved first at MRB 100R or 100F. Retiring of an instruction's result from MRB 100R or 100F to register file 515R or 515F is done when safe, i.e., (a) there is no exception in the execution of the instruction and instructions preceding it, and (b) there is no prior conditional branch instruction that is outstanding or unresolved. This ensures correct execution of a program, giving the same results as if the program was run sequentially. Retiring of a store instruction, which involves a permanent write to D-cache (or memory, I/O device, or other storage unit) 511, follows the same procedure. A summary of the flow of operations involved in each processing stage is depicted in a flowchart shown in FIGS. 35 and 36.

Detailed Description Paragraph Right (187):
A unique retire process, using the branch_point and in_order_point, has been introduced. The branch_point (generated by IAU 560) is the inst_ID of the "oldest" unresolved conditional branch (inst_ID of IAU 560's bottom DIQ 300 or 400 entry). Therefore, all instructions prior to the conditional branch (inst_ID<branch_point) are non-speculative. The in_order_point (generated by IIU 522) is the inst_ID of the "oldest" instruction that has not completed or completed with an exception. Thus, if an instruction has inst_ID <in_order_point, then its preceding instructions completed without an exception. Unlike Johnson's RB and the Metaflow THUNDER SPARC's central DCAF, this retire process using the branch_point and in_order_point eliminates the need for allocating "dummy entries" to the result shelf for branches, stores, or any other instructions that do not write to a register file. It also eases synchronization in multiple result shelves and store buffers.

Detailed Description Paragraph Right (191):
On the performance side, the good characteristics of the RS technique in achieving maximum machine parallelism have been maintained in the DIQ 300 or 400 technique. The only sacrifice made in DIQ 300 technique is the use of in-order issue with an instruction window. This may penalize performance slightly on the cycle count, which can be easily recovered through faster and simpler circuit implementation. In the end, the actual speed or performance of the processor is faster due to reduced cycle time or more operations executed per cycle. (The out-of-order issue DIQ 400 technique is at par with the RS technique in terms of cycle-count performance, but higher in terms of overall performance if the improved clock frequency is factored in.) The performance analysis confirms that a good performance speedup, on the cycle count basis, is still achieved. Based on the benchmark set used, a speedup between

2.6.times. to 3.3.times. was realized in a 4-way superscalar model over its scalar counterpart. Moreover, the performance saturates at a relatively low number of 4 DIQ 300 or 400 entries. These results can be compared to 4-way superscalar processors which typically gain less than 2.0.times. over scalar designs on the SPECint92 benchmarks (see L. Gwennap, "Architects Debate VLIW, Single Chip MP," Microprocessor Report, Vol. 8, No. 16, pp. 20-21, Dec. 5, 1994).

Detailed Description Paragraph Right (197):
FIG. 37 shows the organization of Register Tag Unit (RTU) 523. RTU 523 maintains the most recent tag of every "R" and "F" register (which may be in 515R and 515F or in MRBs 100R and 100F). The most recent tag of a register is the inst_ID of producer instruction that updates the register last. To store tags of all "R" and "F" registers (may be in 515R and 515F or in MRB 100R and 100F), Register Tag File (RTF) 600 is used. RTF 600 has the structure of a register file, except the register content is not a value, but rather a tag. To support speculative execution, Register Tag Reorder Buffer (RTRB) 100RT accompanies RTF 600. RTF 600 are similar to 515R or 515F except that it holds to register values, only register tags, and it was in_order_point to clear the stale/old register tag. (Note that RTRB is not similar to MRB 100.) RTRB 100RT has RTU allocate ports 810, both RTRB 100RT and RTF 600 share read ports 830, and tags are updated from RTRB 100RT to RTF 600 through RTRB retire ports 840. Register tags of source operands RS1(0 . . . N.sub.d -1) and RS2(0 . . . N.sub.d -1) are read from both RTF 600 and RTRB 100RT (see left side of FIG. 37). If a match is found in RTRB, then it contains the most updated register tag, and the tag in RTF is considered old. Tags in RTRB 100RT are updated to RTF 600 as quickly as possible, provided that all previous conditional branches have been resolved. Thus, RTRB 100RT will mostly contain "speculative" register tags. These tags are flushed when the conditional branch is found to be mispredicted. To keep up with the instruction fetch rate, up to N.sub.d entries at the bottom of RTRB 100RT can be simultaneously retired to RTF 600. FIG. 38 shows the structure of RTF 600. RTF cells 695 which include DFFs 605 (or equivalents thereof) are shown in FIG. 38. Also shown in cell 695 is comparison unit 652 (may be an equivalent device to perform the same function as would be understood to those of ordinary skill in the art) for clearing the stale/old register tag. Note that Since there can be multiple updates to the same register location among N.sub.d retirees, a priority selector is accommodated to make sure only the last update is written. Also note that a difference between RTRB 100RT and RTF 600 with regard to speculative tags is that RTRB 100RT holds speculative register tags while RTF 600 holds nonspeculative tags of active instructions (not stale).

Detailed Description Paragraph Right (200):
The fourth modification, read bypassing, is necessary because the most recent tag 740 of a register may still be in allocate ports 810, not yet written to RTRB 100RT. Consider the following example where ((x-y).sup.2 +z).sup.2 computation is about to take place. Assume all calculations are performed in single-precision floating point arithmetics; variables x, y, z were already loaded into register F1, F2, F3, respectively; and the result is written to register F4. Suppose the current decode group is as follows: (N.sub.d =4)

Detailed Description Paragraph Left (3):
Note that a branch instruction is formatted as an integer (floating-point) instruction if its condition code is in an integer (floating-point) register. The DEC ALPHA 21064 avoids condition codes, special registers, or any other single copy of a resource which can potentially become a point of contention in a multi-instruction issue environment. Compare instructions write directly to any general-purpose register (integer or floating-point, depending on the compare operation type).

Detailed Description Paragraph Left (9):
where i.epsilon.[1,N.sub.d -1]. The increment from the base tail pointer 360 is determined based on the number of allocations in the previous ports. The next cycle tail_DIQ(0) 361 is set to (tail_DIQ(N.sub.d -1)+DIQ_alloc_en(N.sub.d -1)) mod N.sub.diq, provided no branch misprediction is detected (mispred_flag=0). If mispred_flag=1, then the next cycle tail_DIQ(0) 361 is set to the DIQ_flush_tail from the DIQ Flush Address Queue (DIQFAQ) 365. This essentially flushes instructions in the mispredicted branch path (if any). DIQFAQ 365 is identical to the one used

MRB 100, providing multi-level speculative execution. Instructions are issued in order from the bottom of DIQ 300, pointed by issue pointer 380 as determined by head (issue) pointer logic 370. Issue pointer 380 is equivalent to a head pointer (issue_DIQ=head_DIQ) and therefore may also be designated as head pointer 380. If there is an instruction (DIQ_empty=0), its register operands are read from the result shelf or directly from the register file. If both reads are successful (valid_read(L)=1 and valid_read(R)=1) then the instruction is issued for execution, and DIQ 300 is popped. The DIQ head pointer 380 is then advanced by one position, ##EQU7##

Detailed Description Paragraph Type 1 (20):
Maximum of one floating-point arithmetic instruction,

Detailed Description Paragraph Type 1 (27):
Many decoded instructions have only one register operand (arithmetic instructions with immediate value operand, loads, conditional branches, floating-point convert instructions), or worse, no register operands at all (jumps, traps).

Detailed Description Paragraph Type 2 (6):
The AXP instruction set includes conditional move instructions for both integer and floating-point data. These instructions should help remove some conditional branches (see section below entitled Condition Move Transformation).

Detailed Description Paragraph Table (1):
Program Example for Reorder Buffer PC Instructions Comments Latency I0: 0 R2 <- 0 ;initialize loop index I1: 1 R0 <- 0 ;initialize loop count I2: 2 R5 <- 1 ;loop increment value I3: 3 R7 <- 100 ;maximum loop count I4: 4 L1:R1 <- (R2 + A) ;load A(I) 11 cycles I5: 5 R3 <- (R2 + B) ;load B(I) 11 cycles I6: 6 R4 <- R1 +.sub.f R3 ;floating-point add 6 cycles I7: 7 R0 <- R0 + R5 ;increment loop count 2 cycles I8: 8 (R0 + C) <- R4 ;store C(I) I9: 9 R2 <- R2 + R5 ;increment loop index 2 cycles I10: 10 P = L1:R0! = R7 ;cond. branch not equal

Detailed Description Paragraph Table (5):
Valid Dual Issue: Instruction A Instruction B integer operate floating-point operate integer/floating-point load integer/floating-point operate/branch integer store integer operate integer store floating-point branch floating-point store floating-point operate floating-point store integer branch integer branch integer operate floating-point branch floating-point operate

Detailed Description Paragraph Table (6):
TABLE 3 Comparisons of Commercial Superscalar Microprocessors Metaflow IBM THUNDER TI DEC HP Intel RS/6000 SPARC SUPERSPARC ALPHA 21064 PA-7100 PENTIUM Integra- Multi-chip Multi-chip Single-chip Single-chip Multi-chip Single-chip tion Clock 62.5 MHz 80 MHz 50 MHZ 200 MHZ 99 MHz 66 MHz Speed (est) SPECint92 61.7 200 (est) 68 116.5 80.0 64.5 SPECfp92 133.2 350 (est) 85 193.6 150.6 56.9 Supersca- All six All six All six, Multi-inst. All six, All six, lar Fea- except re- issue, except re- except re- tures gister decoupled gister re- gister re- renaming dataflow naming and naming sched., speculative out-of-or- execution. der exec. Instructio FIFO I-buf- DCAFs Central Central Central Central n Shelving fers (cen- (DRIS) win-dow and window window window tral, dist. (central, dist. win- (pre-fetch (prefetch (prefetch in FXU and branch, dow (for FP buf-fers) buffers) buffers) FPU) floating- inst.) point) Result Reg.-map- DCAFs None None None None Shelving ping table in FPU Indepen- 1 branch 1 branch 1 branch 1 branch 1 integer 2 ALUs, dent unit, unit, unit, unit, unit (ALU, 1 FP unit Execution 1 FX unit, 3 ALUs, 3 ALUs, 1 address shift, (add, muit, Units 1 FP unit 1 FP add, 1 address unit, branch div) (MAF) 1 FP mult add, 1 integer add), 1 FP unit unit, 1 FP unit (add, mult) 1 FP unit (mult, (add, mult, div/sqrt) div) Decode 4 instruc- 4 instruc- 4 instruc- 2 instruc- 2 instruc- 2 instruc- Size tions tions tions tions tions tions Max Issue 1 FX or FP 1 branch Triple Dual issue Dual issue Dual issue load/store inst., issue with of certain of integer of "simple" inst., 2 integer certain integer/ and floa- instruc- 1 FP arith. inst., restriction floating- ting-point tions. inst., 1 load/ s. point instruc- 1 branch store inst, operate, tions. inst., 1 FP add/ branch, and 1 condi- sub, load/store. tion-regi- 1 FP ster inst. multiply Branch Static Dynamic Static (al- Static and Static Dynamic Prediction (con-stant ways taken) dynamic (BTFN) pre-dicted- not-taken) Notes .cndot. FP mult- .cndot. Has the .cndot. Multiple-

.cndot. Hybrid of .cndot. Uses off- .cndot. Supports add in 2 most com- path fetch- superpipe- chip, pri- the wide- cycles. plete dyna- ing into lined and mary caches ly-used x86 .cndot. This mic hard- se-quential supersca- for size inst. set. RS/6000 de- ware sche- and target lar. and speed .cndot. The only sign is the duler with inst. .cndot. True 64- flexi- superscalar foundation full out- queues bit archi- bility processor of of-order helps tecture. .cndot. Supports that is not follow-on issue. reduce .cndot. Supports VLIW-like a register- single-chip .cndot. Low clock branch cond. move and SIMD- to-regi- versions speed due mispredic- inst. like inst. ster, 3-ad- (PowerPC to complex tion .cndot. Supports for path- dress ma- 601, 603, out-of-or- penalty. multiple length chine. 604, 620) der issue. O/S using reduction. .cndot. .cndot. Precise .cndot. Thunder PALcode. Inefficient interrupts SPARC was a .cndot. Imprecise inst. only in reborn of interrupts. window due "synchroniz the to vari- e" mode. unsuccessfu able - length l Lightning x86 inst. SPARC.

Current US Original Classification (1):
712/23

CLAIMS:

7. The reorder buffer system as claimed in claim 4, wherein said reorder buffer system is connected to a register file by a retire bus, and wherein said reorder buffer system retires information to said register file over said retire bus.

L11: Entry 3 of 3 File: USPT Oct 20, 1998

DOCUMENT-IDENTIFIER: US 5826096 A
TITLE: Minimal instruction set computer architecture and multiple instruction issue method

Abstract Paragraph Left (1):
A minimal instruction set computer architecture (hyperscalar computer architecture) comprises a central memory, an instruction buffer, a control unit, an I/O control unit, a plurality of functional units, a plurality of register files, and a data router. In the hyperscalar computer architecture, the central memory transfers a plurality of instructions to the instruction buffer. The control unit receives multiple instructions from the instruction buffer, and automatically determines and issues the largest subset of instructions from those received that can be simultaneously issued to the plurality of functional units. Each functional unit receives data from and returns computational results to a corresponding register file. The data router serves to transfer data between each register file and any other register file, the central memory, the control unit, or the I/O control unit. The present invention also includes a multiple instruction issue method for issuing instructions to the hyperscalar computer architecture. The multiple instruction issue method comprises the steps of: determining a set of first source register files used by a plurality of instructions; determining a set of second source register files used by the plurality of instructions; determining a set of destination register files used by the plurality of instructions; determining a largest subset of instructions within the plurality of instructions that can be executed without a register file conflict; and issuing in parallel each instruction within the largest subset to the plurality of functional units.

Brief Summary Paragraph Right (5):
Reduced Instruction Set Computing (RISC) architectures attempt to meet this need by increasing the speed at which each computational instruction supported is performed. RISC is based upon the premise that a complex computational instruction can typically be performed through a corresponding sequence of simple instructions. The smallest number of simple instructions required to fully implement a complete instruction set are chosen as the RISC instruction set. A given RISC architecture is designed to execute each simple instruction in its instruction set very quickly, such that a given complex instruction can be executed very rapidly as a corresponding sequence of simple instructions. A block diagram of an exemplary RISC architecture is shown in FIG. 1. This architecture comprises a memory, a data cache, an instruction cache, pipelined instruction decoding, a register file, a floating-point functional unit, and an integer functional unit. In the RISC architecture, either the integer functional unit or the floating-point functional unit can be used for a given computation.

Brief Summary Paragraph Right (6):
All data transferred between the register file and the memory in the RISC architecture must pass through the data cache. The data cache serves as a high-speed, small capacity memory. A data cache size of 16 kilobytes is currently common (1993). Hence, in a data-intensive computational environment such as image processing, the data cache can store only a very small portion of the total amount of data that must be processed. Presently, a backing cache is occasionally used between a CPU's internal cache and memory to provide additional cache storage. In the future, cache sizes will increase as improvements are made in microelectronic processing technology. Regardless of the cache situation (present or future), a cache serves as an information "bottleneck" between the memory and the register file. This limits the overall processing speed of the RISC architecture, particularly in data-intensive applications. Therefore, RISC architectures are not

well-suited for processing large amounts of data, such as in image or document processing, since access to any data element stored within memory necessitates routing the data element through the data cache.

Brief Summary Paragraph Right (11):
The present invention is a minimal instruction set computer architecture and multiple instruction issue method for processing multiple scalar instructions in parallel using a minimal instruction set. The computer architecture is referred to herein as a hyperscalar computer architecture. The hyperscalar computer architecture preferably comprises a central memory, an instruction buffer, a control unit, an I/O control unit, a plurality of functional units, a plurality of register files, and a data routing circuit. In the hyperscalar computer architecture, the central memory transfers a plurality of instructions to the instruction buffer. The control unit receives multiple instructions from the instruction buffer, and automatically determines a largest subset of instructions from those received that can be simultaneously issued to the plurality of functional units. The control unit then issues each instruction in the largest subset in parallel to an appropriate functional unit. Each functional unit receives data from and returns computational results to an associated register file. The data routing circuit serves to transfer data between each register file and any other register file, the central memory, the control unit, or the I/O control unit.

Brief Summary Paragraph Right (12):
The preferred embodiment of the multiple instruction issue method comprises the steps of: determining a set of first source register files used by a plurality of instructions; determining a set of second source register files used by the plurality of instructions; determining a set of destination register files used by the plurality of instructions; determining a largest subset of instructions within the plurality of instructions that can be executed without a register file conflict; and issuing in parallel the instructions within the largest subset to a plurality of functional units.

Detailed Description Paragraph Right (1):
Referring now to FIG. 2, a block diagram of a preferred embodiment of a hyperscalar computer architecture 10 constructed in accordance with the present invention is shown. The hyperscalar computer architecture 10 preferably comprises a central memory 100, an instruction buffer 120, a control unit 140, an I/O control unit 130, a plurality of functional units 270, a plurality of register files 260, and a data routing circuit 300. The instruction buffer 120 receives multiple instructions in parallel from the central memory 100. The control unit 140 in turn receives a plurality of instructions from the instruction buffer 120, and automatically determines the largest sequential number of instructions within the plurality of instructions received from the instruction buffer 120 that can be issued in parallel to the plurality of functional units 270. Since the control unit 140 automatically detects instruction sequences that can be processed in parallel, a specialized compiler for arranging serial instructions into parallel sequences is unnecessary. The data routing circuit 300 serves to transfer data to and from the central memory 100, the control unit 140, the I/O control unit 130, and each register file 260.

Detailed Description Paragraph Right (2):
In the preferred embodiment of the present invention, the data word size is 64 bits. This data word size is a natural choice for image processing applications, since each pixel within an image can be represented by four 16-bit values corresponding to red, green, blue, and opacity values. The 64-bit word size is also a natural choice for document processing applications, since each pixel in a document can be represented by four 16-bit values corresponding to cyan, yellow, magenta, and black. The 64-bit data word size is also the natural data word size for performing double precision floating point operations.

Detailed Description Paragraph Right (3):
The hyperscalar computer architecture 10 is a "three address" architecture, in that instructions specify a first location within a first source register file 260 from which to obtain data, a second location within a second source register file 260 from which to obtain data, and a third location in a result destination register file 260 at which to store the result of the operation performed by the instruction.

Each instruction also specifies an opcode indicating the specific operation the instruction is to perform. Each instruction therefore can be represented as I=(op(i,j,k)), where "op" is the opcode, i identifies the result destination register file 260 and an address within the result destination register file 260; j identifies the first source register file 260 and an address within the first source register file 260; and k identifies the second source register file 260 and an address within the second source register file 260. In the preferred embodiment, the instruction word size is 32 bits.

Detailed Description Paragraph Right (4):
The central memory 100 preferably comprises a plurality of interleaved memory banks having an instruction output, an address input, a control input, and a bi-directional data cluster port. Upon receiving an appropriate control signal at its control input, the central memory 100 transfers a first instruction group to the instruction output. In the preferred embodiment, the first instruction group size is 16 instructions. An address received at the address input specifies the starting address at which to begin a first instruction group transfer. The data cluster port comprises a number of data pathways equal to the number of register files 260, which in the preferred embodiment is eight. Each data pathway within the data cluster port has a width equal to the data word size, which in the preferred embodiment is 64 bits. Thus, in the preferred embodiment, the data cluster port is 8*64 or 512 bits wide. In an exemplary embodiment, the central memory 100 is 8 banks of interleaved 16 Mb DRAMs.

Detailed Description Paragraph Right (6):
The control unit 140 is a means for issuing multiple instructions in parallel to the plurality of functional units 270. The control unit 140 provides an instruction input, an IB control output, a CM control output, a DRC control output, a CM address output, a plurality of FMAP outputs, and an address I/O (AIO) port. The instruction input of the control unit 140 is coupled to the instruction output of the instruction buffer 120 by a second instruction transfer path 14. The IB control output of the control unit 140 is coupled to the instruction buffer's control input via a control line 16. An address line 18 couples the control unit's CM address output and the central memory's address input. A control line 15 couples the CM control output to the control input of the central memory 100. The control unit 140 directs the instruction buffer 120 to transfer the issue consideration group via the IB control output. Through the CM control output, the control unit 140 directs the central memory 100 to transfer one or more first instruction groups to the instruction buffer 120, where the starting address of the first instruction group is output at the control unit's CM address output. Upon receiving the instructions within the issue consideration group from the instruction buffer 120, the control unit 140 determines the largest subset of instructions within the issue consideration group that can be issued in parallel in a single clock cycle. The control unit 140 then issues each of the instructions within the largest subset to an appropriate functional unit 270 by directing a function map, or FMAP, to a corresponding FMAP output. Each FMAP contains an instruction opcode, addresses for obtaining a first source data value, a second source data value, and a result destination address. The control unit 140 maintains a current instruction pointer (CIP) indicating the address of the first instruction that has been issued to the plurality of functional units 270 at the outset of the current clock cycle; a current issue count signal Y that indicates the number of instructions issued at the outset of the current clock cycle; a next instruction pointer (NIP) that indicates the address of the first instruction to be issued to the plurality of functional units 270 in the next clock cycle; and a next issue count signal P that indicates the number of instructions to be issued at the outset of the next clock cycle.

Detailed Description Paragraph Right (7):
Each functional unit 270 comprises a means for performing logical and arithmetic operations. Each functional unit 270 has a first input, a second input, an FMAP input, and an output. The FMAP input of each functional unit 270 is coupled to a corresponding FMAP output of the control unit 100 by an FMAP transfer path 20. Under the direction of the control unit 140, each functional unit 270 receives an FMAP corresponding to an instruction at its FMAP input. Each FMAP is obtained from a portion of an instruction and indicates the address of a first data item in the first source register file 260, the address of a second data item in the second

source register file 260, an opcode corresponding to the operation to be performed on the first and second data items respectively stored at the first and second source register file 260 addresses, and an address in the result destination register file 260 at which to store the result of the operation. The opcode within the FMAP indicates the address of a microprogram that is executed by the functional unit 270 to carry out the actions necessary to implement the instruction.

Detailed Description Paragraph Right (8):
Each register file 260 comprises a data storage means for storing a plurality of data words. Each register file 260 has a first input, a second input, and a first output, and a second output. The first and second outputs of each register file 260 are coupled to the first and second inputs of a corresponding functional unit 270 via a first data line 30 and a second data line 32, respectively. The second input of each register file 260 is coupled to the output of its corresponding functional unit 270 via a result line 34, and therefore, receives the result of an operation performed by its corresponding functional unit 270 at the end of the current clock cycle. In the preferred embodiment, each register file 260 stores 64 data words.

Detailed Description Paragraph Right (10):
The data routing circuit 300 comprises a means for selectively routing data between a given register file 260 and any other register file 260, the central memory 100, the control unit 140, or the I/O control unit 130. The data routing circuit 300 has a control input, a bi-directional data cluster port, a bi-directional AIO port, a bi-directional IO port, and a plurality of bi-directional data word ports. The control input of the data routing circuit 300 is coupled to the DRC control output of the control unit 140 by a control signal bus 17. The data routing circuit's data cluster port is coupled to the central memory's data cluster port by a cluster bus 22. Each data word port is coupled to the first input of a respective register file 260 by a data word bus 28. The AIO port of the data routing circuit 300 is coupled to the AIO port of the control unit 140 by an AIO bus 24, and the data routing circuit's IO port is coupled to the IO port of the I/O control unit 130 by a IO bus 26. The cluster bus 22 comprises a number of data pathways equal to the number of register files 260 present in the hyperscalar computer architecture 10, where each data pathway is one data word wide. Since each data pathway carries a data word, the signal present on the cluster bus 22 is a data word cluster. In the preferred embodiment, the cluster bus 22 is eight 64-bit data pathways, having a total width of 512 bits. The data routing circuit 300 allows each data pathway within the cluster bus 22 to be routed to a unique register file 260. Thus, if a data word is present on each of the data pathways, each data word can be routed to a unique register file 260 or to a unique location in central memory 100. Moreover, a data word can be routed between any register file 260 and the AIO port of the control unit 140 or the IO port of the I/O controller 130. The specific routing performed by the data routing circuit 300 is determined by a signal received at the data routing circuit's control input via the control signal bus 17.

Detailed Description Paragraph Right (11):
The hyperscalar computer architecture 10 is a scalable computer architecture in that the maximum number of instructions that can be issued in parallel is given by a scaling parameter K, where K equals 2.sup.M. The scaling parameter is chosen based upon the hardware resources available. Since the scaling parameter K determines the maximum number of instructions that can be issued in parallel, it also determines the number of functional units 270 that must be present, which in turn determines the number of register files 260 that are required. To allow parallel transfer of data between the plurality of register files 260 and the central memory 100, the number of data word buses 28 must match the number of register files 260, and the cluster bus 22 must have a width equal to the number of register files 260 times the data word size.

Detailed Description Paragraph Right (12):
The choice of scaling parameter also determines the number of bits required in an instruction to uniquely specify the first source register file 260, the second source register file 260, and the result destination register file 260. The choice of scaling parameter therefore also determines the number of bits within an instruction that are available to specify the instruction opcode and the unique addresses of the first data item, the second data item, and the instruction result

within the first source, the second source, and the result destination register files 260, respectively.

Detailed Description Paragraph Right (13):
In the preferred embodiment of the present invention, the scaling parameter is 8. Therefore, up to eight instructions can be issued by the control unit 140 in parallel to a total of eight functional units 270. Eight register files 260, each register file 260 corresponding to a functional unit 270, are present. Eight data word buses 28 couple each register file 260 to the data routing circuit 300, and the cluster bus 22 is eight data words or 512 bits wide. Within each 32-bit instruction, three bits are required to indicate the first source register file 260, three bits are required to indicate the second source register file 260, and three bits are required to indicate the result destination register file 260. Thus, a total of 9 instruction bits are required to identify the register files 260 indicated in the instruction as a result of the scaling parameter having the value 8.

Detailed Description Paragraph Right (14):
Since each register file 260 in the preferred embodiment can store a total of 64 data words, each instruction requires 6 bits to indicate a specific address within each of the first source, second source, and result destination register files 260. Therefore, a total of 18 instruction bits are required to indicate register file addresses. The 9 instruction bits required to identify the first source, the second source, and the result destination register files 260 plus the 18 instruction bits required to identify addresses within each of these register files 260 leaves 5 instruction bits available to specify an instruction opcode within the 32-bit instruction size of the preferred embodiment. This in turn means that a total of 32 instructions are supported in the hyperscalar computer architecture's instruction set. Generally, commercially available RISC instruction sets comprise a variable number of instructions, depending upon the particular RISC architecture. However, some RISC instruction sets comprise more than one hundred instructions. The hyperscalar computer architecture 10 is therefore a minimal instruction set computing (MISC) architecture. The instruction set for the preferred embodiment 10 of the hyperscalar computer architecture is given in Appendix A.

Detailed Description Paragraph Right (16):
In the representation of each instruction as I=(op(i,j,k)), the instruction bits within i, j, and k indicate the result destination register file 260 and an address within the result destination register file 260; the first source register file 260 and an address within the first source register file 260; and the second source register file 260 and an address within the second source register file 260, respectively. The number of register files 260 present is given by the scaling parameter K, where K equals 2.sup.M. Therefore, M bits are required within each of i, j, and k to specify a particular register file 260. Herein, the M bits within i are labeled as T, the M bits within j are labeled as R, and the M bits within k are labeled as S. Thus, within each instruction, R indicates the first source register file 260, S indicates the second source register file 260, and T indicates the result destination register file 260. Therefore, the register files 260 indicated by each instruction in the issue consideration group are given by a bit vector T.sub.v R.sub.v S.sub.v, where v ranges from 0 to (2.sup.M -1). The TRS bit array is therefore formed from the bit vector T.sub.v R.sub.v S.sub.v created from each instruction within the issue consideration group.

Detailed Description Paragraph Right (17):
Within the control unit 140, the TRS bit array is used to determine the largest sequential number of instructions from within the issue consideration group that can be issued simultaneously. This number is determined in relation to a first parallel issue constraint and a second parallel issue constraint. The first parallel issue constraint is that the first data item and the second data item must be stored within the register file 260 that is to receive the instruction result. This is required because each functional unit 270 operates on data that is stored within its corresponding register file 260, and returns a result to its corresponding register file 260 upon completion of the operation. If the first data item or the second data item are not stored within the result destination register file 260, a data transfer operation from the first source register file 260 and/or a data transfer from the second source register file 260 to the result destination register file 260 are/is

required. This data transfer operation must occur through the data routing circuit 300. Since instructions are issued in parallel, but program serialism must be maintained for the coherency in register side-effecting, a data transfer operation precludes the use of the functional units 270 corresponding to the register files 260 involved in the data transfer operation until the transfer operation has been completed.

Detailed Description Paragraph Right (18):
The second parallel issue constraint that must be satisfied for two or more instructions to be issued in parallel is that each instruction issued must specify a unique result destination register file 260 to avoid a register file usage conflict. For example, if the first instruction within the issue consideration group specifies a data transfer operation to a given register file 260, and the fifth instruction within the issue consideration group indicates that the same register file 260 is to be the recipient of a result generated by its corresponding functional unit 270, the first instruction and the fifth instruction cannot be issued in parallel.

Detailed Description Paragraph Right (20):
The FIJK bit array is created from those instruction bits that are not used in creating the TRS bit array. In a manner similar to the TRS bit array, the FIJK bit array is created from a plurality of F.sub.v I.sub.v J.sub.v K.sub.v bit vectors, where v ranges from 0 to (2.sup.M -1). Within the FIJK bit array, each F.sub.v indicates the opcode for instruction v; each I.sub.v indicates the address within the result destination register file 260 for instruction v; each J.sub.v indicates the address of the data item within the first source register file 260 for instruction v; and each K.sub.v indicates the address of the data item within the second source register file 260 for instruction v. Each F.sub.v I.sub.v J.sub.v K.sub.v bit vector is a portion of an FMAP that is routed to one of the functional units 270, indicating the operation to be performed, the addresses of the data items upon which to operate, and the address at which the result of the operation is to be stored.

Detailed Description Paragraph Right (29):
The branch detection circuit 180 comprises a decoding means for determining if an instruction corresponding to a branch is within the issue consideration group, and has an input, a branch condition output, and a branch output. The input of the branch detection circuit 180 is coupled to the FIJK bus 144. The branch condition output of the branch detection circuit 180 is coupled to the branch condition input of the control unit address bitslice 230 via a branch condition line 184. The branch output of the branch detection circuit is coupled to the branch input of the microsequencer 220 via a branch line 182. The branch detection circuit 180 decodes the F values present within the FIJK bit array. If an F value corresponding to a branch is detected, the branch detection circuit 180 outputs a branch condition signal to the control unit address bitslice 230, and a branch signal to the microsequencer 220. In a branch instruction I=(op,(i,j,k)), the i portion of the instruction is used to indicate a register file address, the contents of which has a value to be tested for a given condition depending upon the instruction opcode. The branch detection circuit 180 includes in the branch condition signal the particular test that is required.

Detailed Description Paragraph Right (31):
The first parallel issue constraint circuit 212 and the second parallel issue constraint circuit 218 each have a respective input and a respective output. The input of the first parallel issue constraint circuit 212 and the input of the second parallel issue constraint circuit 218 are coupled to the second TRS bus 146. The issue count selection circuit 216 has a first input, a second input, and an output. The first input of the issue count selection circuit 216 is coupled to the output of the first parallel issue constraint circuit 212, and the second input of the issue count selection circuit 216 is coupled to the output of the second parallel issue constraint circuit 218. The output of the issue count selection circuit is coupled to the issue count line 148. The first parallel issue constraint circuit 212 produces a first issue count at its output that indicates the largest sequential number of instructions that satisfy the first parallel issue constraint, namely, that the result destination register of each instruction that is issued in parallel must be unique. The second parallel issue constraint circuit 218 produces a second

issue count at its output that indicates the largest sequential number of instructions that satisfy the second parallel issue constraint. The second parallel issue constraint is that for a given bit vector T.sub.v R.sub.v S.sub.v, a register file conflict exists if R.sub.v does not equal T.sub.v or if S.sub.v does not equal T.sub.v. In such cases, either the first data item or the second data item must be retrieved from a register file 260 other than the result destination register file 260. The issue count selection circuit 216 outputs the smaller of the first issue count and the second issue count as the next issue number P.

Detailed Description Paragraph Right (32):
Referring now to FIGS. 10A, 10B, and 10C, a block diagram of a preferred embodiment of the first parallel issue constraint circuit 212 is shown. The first parallel issue constraint circuit 212 comprises a plurality of comparators 500, a plurality of OR gates 502, 503, 504, 505, 506, 507, a plurality of AND gates 510, a plurality of inverters 511, and a first priority encoder 550. Each comparator 500 has a first input, a second input, and an output. To satisfy the first parallel issue constraint, result destination register file 260 conflicts must be detected between each T.sub.v within the TRS bit array and T.sub.v-1, T.sub.v-2, T.sub.v-3, and so on to T.sub.0. Thus, each T.sub.v is compared with T.sub.v-1, T.sub.v-2, T.sub.v-3, and so on to T.sub.0, beginning with T.sub.1. Therefore, for each given T.sub.v, where v>0, there are v comparisons required, necessitating the use of v comparators 500. In the preferred embodiment, v ranges from 0 to 7. Hence, in the preferred embodiment, seven comparators 500 are required to compare T.sub.7 to T.sub.0 through T.sub.6 to indicate a T.sub.7 conflict; six comparators 500 are required to compare T.sub.6 to T.sub.0 through T.sub.5 to indicate a T.sub.6 conflict; five comparators 500 are required to compare T.sub.5 to T.sub.0 through T.sub.4 to indicate a T.sub.5 conflict; four comparators 500 to compare T.sub.4 to T.sub.0 through T.sub.3 to indicate a T.sub.4 conflict; three comparators 500 are required to compare T.sub.3 to T.sub.2, T.sub.1, and T.sub.0 to indicate a T.sub.3 conflict; two comparators 500 are required to compare T.sub.2 to T.sub.1 and T.sub.0 to indicate a T.sub.2 conflict; and a single comparator 500 is required to compare T.sub.1 to T.sub.0 to indicate a T.sub.1 conflict. In the preferred embodiment, a total of 28 comparators 500 are required.

Detailed Description Paragraph Right (34):
Within the plurality of OR gates 502, 503, 504, 505, 506, 507, there is an OR gate associated with each T.sub.v comparison having v inputs and an output. For example, the output of each comparator 500 used in the T.sub.7 conflict comparisons is coupled to a corresponding input of a T.sub.7 OR gate 507. Similarly, the output of each comparator 500 used in the T.sub.6 conflict comparisons is coupled to a corresponding input of a T.sub.6 OR gate 506, and so on in a similar manner for a T.sub.5 OR gate 503, a T.sub.4 OR gate 504, a T.sub.3 OR gate 503, and a T.sub.2 OR gate 502. Those skilled in the art will recognize that a T.sub.1 OR gate is not necessary because the T.sub.1 conflict comparison requires only one comparator. A high signal value present at the output of any OR gate within the plurality of OR gates 502, 503, 504, 505, 506, 507 indicates that a result destination register file conflict has occurred.

Detailed Description Paragraph Right (35):
From the plurality of AND gates 510 and the plurality of inverters 511, one AND gate 510 and one inverter 511 corresponds to each value of v greater than one. Each AND gate 510 has a first input and a second input, and each inverter 511 has an input and an output. The first input of each AND gate 510 is coupled to the output of a corresponding OR gate 502, 503, 504, 505, 506, 507. The second input of each AND gate 510 is coupled to the output of a corresponding inverter 511. For a given value of v, where v>1, the input of the inverter 511 corresponding to v is coupled to the output of the AND gate 510 corresponding to (v-1). Since no AND gate 510 corresponding to v=1 is present, the input of the inverter 511 corresponding to v=2 is coupled to the output of the T.sub.1 conflict comparator. Moreover, the final determination of whether eight instructions can be performed is determined by logically combining the inverted output of the T.sub.7 OR gate 507 with the inverted output of the T.sub.7 AND gate 510 using an AND operation. If a multiple result destination register file 260 or T.sub.v conflicts exist, the result destination register file 260 conflict associated with the smallest v determines how many instructions satisfying the first parallel issue constraint can be issued in

parallel. In the presence of multiple T.sub.v conflicts, the output of the AND gate 510 corresponding to the smallest value of v for which a conflict has been detected indicates a conflict while the other AND gate outputs do not indicate a conflict.

Detailed Description Paragraph Right (39):
Within the first comparator set 601, the first comparator determines if a register file conflict exists between the result destination register file T.sub.1 and the first source register file R.sub.1. In like manner, the second comparator within the first comparator set 601 determines if a register file 260 conflict exists between the result destination register file T.sub.1 and the second source register file S.sub.1. The outputs of the first and second comparators 621, 622 are asserted in an inverted state. Thus, if the first comparator 621 has determined that a register file 260 conflict exists, the signal appearing at the first comparator set's first output will have a high value. Similarly, the signal appearing at the first comparator set's second output will have a high value if the second comparator 622 has determined that a register file conflict exists. The first and second comparators 621, 622 within the second through eighth comparator sets 602, 603, 604, 605, 606, 607, 608 function in an analogous manner.

Detailed Description Paragraph Right (40):
The plurality of OR gates 640 in the second parallel issue constraint circuit 218 are each associated with a comparator set 601, 602, 603, 604, 605, 606, 607, 608. Therefore, in the preferred embodiment, eight OR gates 640 are present. Each OR gate 640 has a first input, a second input, and an output. The first input of each OR gate 640 is coupled to its respective comparator set's first output, and the second input of each OR gate 640 is coupled to its respective comparator's second output. If a register file conflict has been indicated by the first comparator 621 or the second comparator 622 within a given comparator set 601, 602, 603, 604, 604, 606, 607, 608, the associated OR gate 640 outputs a signal having a high value at its output.

Detailed Description Paragraph Right (41):
The plurality of AND gates 645 comprises an AND gate 645 corresponding to comparator sets two through eight 602, 603, 604, 605, 606, 607, 608. Thus, in the preferred embodiment, seven AND gates 645 are present. Each AND gate 645 has a first input, a second input, and an output. For the AND gate 645 associated with the eighth comparator set 602, the first input of the AND gate 645 is coupled to the output of the OR gate 640 associated with the eighth comparator set 602. The second input of this AND gate 645 is coupled to the output of the AND gate 645 associated with the seventh comparator set 601. For the AND gate 645 associated with the seventh comparator set 602, the first input of this AND gate 645 is coupled to the output of the OR gate 640 associated with the seventh comparator set 602, and the second input of this AND gate 645 is coupled to the output of the AND gate 645 associated with the sixth comparator set 601. Thus, for each AND gate 645 associated with a given comparator set C, the first input of the AND gate 645 is coupled to the output of the OR gate 640 associated with comparator set C, and the second input of the AND gate 645 is coupled to the output of the AND gate 645 associated with comparator set (C-1), for comparator sets two through eight 602, 603, 604, 605, 606, 607, 608. The signal present at each AND gate's second input is inverted upon arriving at the second input. In the event that more than one register file conflict is detected by the second parallel issue constraint circuit 218, the plurality of AND gates 645 ensures that the register file conflict corresponding to the smallest value of v is used to indicate the number of sequential instructions that satisfy the second parallel issue constraint.

Detailed Description Paragraph Right (44):
Referring now to FIG. 5, a block diagram of a preferred embodiment of the microsequencer 220 within the control unit 140 is shown. The microsequencer 220 comprises an F register 802, an I register 804, a J register 806, and a K register 808 for receiving an external FMAP; a first function register 810 and a second function register 812; a first and a second multiplexor 814, 816; a first and a second register file port 818, 820; a TMAP ROM 822, a T counter 824, a T microcode ROM 826, and a T microinstruction register; an SMAP ROM 822, an S counter 824, an S microcode ROM 826, and an S microinstruction register; an XMAP ROM 822, an X counter 824, an X microcode ROM 826, and an X microinstruction register; a YMAP ROM 822, a Y

counter 824, a Y microcode ROM 826, and a Y microinstruction register.

Detailed Description Paragraph Right (45):
The F, I, J, and K registers 802, 804, 806, 808 each comprise a data storage means having an input, a load input, and an output. The inputs of the F, I, J, and K registers are coupled to the control unit's AIO port such that they receive the F value, the I value, the J value, and the K value of an external FMAP, respectively. In addition, the inputs of the F, I, J, and K registers are each coupled to the branch line 182. The control inputs of the F, I, J, and K registers each receive the L value within the external FMAP. The external FMAP originates from the I/O control unit 130, and is transferred to the microsequencer 220 within the control unit 140 via the AIO bus 24. Those skilled in the art will recognize that a direct coupling from the I/O control unit 130 could also be used to transfer the external FMAP to the control unit's microsequencer 220. As in the case of the FMAPs sent to the plurality of functional units 270, the external FMAP specifies a set of operations that are to be performed. Receipt of the branch signal via the branch line 182 indicates a set of operations that are to be performed in the event that a branch is detected. The L value received from the external FMAP indicates whether the FMAP is to be loaded into the microsequencer 220. The microsequencer 220 carries out the operations required to perform a function indicated by the F value within the external FMAP received. Functions indicated by the F code within the external FMAP include a start continuous execution function, a single-step execution function, and a halt function.

Detailed Description Paragraph Right (49):
The TMAP ROM 822, the T counter 824, the T microcode ROM 826, and the T microinstruction register 828 function together as a master nanosequencer within the control unit's microsequencer 220. The TMAP ROM 822 comprises a read-only memory having an input and an output. The input of the TMAP ROM is coupled to the output of the first function register 810, and therefore receives the F value. The TMAP ROM serves as a lookup table to transform the F value into a start-up address for an appropriate microcode sequence. The T counter 824 comprises a counting means having an input, a control input, and an output. The input of the T counter is coupled to the output of the TMAP ROM, and therefore receives a microcode sequence start-up address. The T counter increments or decrements the start-up address received, thereby tracking addresses within the microcode sequence. The T microcode ROM 826 comprises a read-only memory having an input and an output. The input of the T microcode ROM 826 is coupled to the output of the T counter 824. The T microcode ROM 826 receives the address supplied by the T counter 824, and outputs a corresponding microcode instruction. The T microinstruction register 828 comprises a data storage means having an input, a control input, and an output. The input of the T microinstruction register 828 is coupled to the output of the T microcode ROM. Therefore, the T microinstruction register 828 receives a microcode instruction. The master nanosequencer serves to control overall operation sequencing for the first, second, and third subordinate nanosequencers, as well as for the first and second multiplexors 814, 816 and the first and second output registers 818, 820. The output of the T microinstruction register is used to supply overall timing signals required within the control unit's microsequencer 220.

Detailed Description Paragraph Right (50):
The SMAP ROM 832, the S counter 834, the S microcode ROM 836, and the S microinstruction register 838 form a first subordinate nanosequencer. The SMAP ROM 832 comprises a read-only memory having an input and an output. The input of the SMAP ROM is coupled to the output of the second function register 812, and therefore receives the F value. The SMAP ROM serves as a lookup table to transform the F value into a start-up address for an appropriate microcode sequence. The S counter 834 comprises a counting means having an input, a control input, and an output. The input of the S counter is coupled to the output of the SMAP ROM, and therefore receives a microcode sequence start-up address. The S counter increments or decrements the start-up address received, thereby tracking addresses within the microcode sequence. The S microcode ROM 836 comprises a read-only memory having an input and an output. The input of the S microcode ROM 836 is coupled to the output of the S counter 824. The S microcode ROM 836 receives the address supplied by the S counter 834, and outputs a corresponding microcode instruction. The S microinstruction register 838 comprises a data storage means having an input, a

control input, and an output. The input of the S microinstruction register 838 is coupled to the output of the S microcode ROM. Therefore, the S microinstruction register 838 receives a microcode instruction. The microinstruction received is output at the S microinstruction register's output.

Detailed Description Paragraph Right (51):
The XMAP ROM 842, the X counter 844, the X microcode ROM 846, and the X microinstruction register 848 form a second subordinate nanosequencer. The YMAP ROM 852, the Y counter 854, the Y microcode ROM 856, and the Y microinstruction register 858 form a second subordinate nanosequencer. Those skilled in the art will recognize that the each element's couplings and functionality within the second and third subordinate nanosequencers are analogous to those for the elements in the first subordinate nanosequencer. The outputs of the S microinstruction register 838, the X microinstruction register 848, and the Y microinstruction register 858 supply the signals required for instruction buffer 120 control and central memory 100 control.

Detailed Description Paragraph Right (57):
The plurality of buffers 390 comprises a plurality of data storage means, where each buffer 390 within the plurality has a first bidirectional data word port and a second bidirectional data word port. Each first bidirectional data word port within the plurality of buffers 390 is coupled to a corresponding data word output of the first switch array 370. Each second bidirectional data word port within the plurality of buffers 390 forms a corresponding bidirectional data word port of the data routing circuit 300. Each buffer 390 therefore serves to transfer data between a corresponding register file 260 and the first switch array 370.

Detailed Description Paragraph Right (59):
In the preferred embodiment of the hyperscalar computer architecture 10, 2.sup.M data words occupy a data word cluster. Each data word output within the first switch array's plurality of data word outputs corresponds to a predetermined register file 260. Similarly, each data word input within the second switch array's plurality of data word inputs corresponds to a predetermined register file 260. Since 2.sup.M register files 260 are present in the preferred embodiment of the hyperscalar computer architecture, the plurality of data word outputs of the first switch array 370 and the plurality of data inputs of the second switch array 375 can be considered to be a data word cluster. Therefore, in the preferred embodiment of the hyperscalar computer architecture 10, the first switch array 370 and the second switch array 375 are identical elements.

Detailed Description Paragraph Right (64):
The data routing circuit 300 operates in a "forward" transfer mode, defined as a data word cluster transfer to the plurality of register files 260, or in "reverse" transfer mode when a data word cluster is transferred from the plurality of register files 260 to another location. In the forward transfer mode, a data word cluster can be routed from the cluster bus 22 to the plurality of register files 260, where each data word within the data word cluster is uniquely routed to a given register file 260. The particular routing applied to each data word in the data cluster is determined by the signal present on the third select line 44, and is performed by the first switch array 370. Alternatively, a single data word can be routed from either the AIO bus 24 or the IO bus 26 to each register file 260 within the plurality of register files 260. When a single data word is routed in the forward transfer mode, the data word duplication line 310 inserts the single data word into each data word of a data word cluster. In this case, each data word within the data word cluster is identical to the single data word.

Detailed Description Paragraph Right (74):
Referring now to FIG. 8, a block diagram of a preferred embodiment of each functional unit 270 and register file 260 is shown. Each functional unit 270 and register file 260 comprises a FU microsequencer 272, an integer processing/register file unit 271, a multiply/divide unit 276, an add/subtract unit 274, and a floating point register 278. The integer processing/register file unit 271 comprises 64-bit integer arithmetic hardware and the register file 260 associated with the functional unit 270. The integer processing/register file unit 271 has an I input, an A input, a B input, a control input, a data word input, and a data word output. The data word input of the integer processing/register file unit 271 is coupled to its data word

output, to the data word bus 28 corresponding to the register file 260, and to a first 64-bit line 281. The signal present at the I input indicates the address at which to store the result of a computation performed by the functional unit 270. The A and B inputs receive the addresses of the first and second data items. The integer processing/register file unit's control input specifies a specific integer operation or a register file data transfer operation. The result of an integer operation or the data item associated with a data transfer operation appears at the integer processing/register file unit's data word input and data word output. A signal received at the integer processing unit's control input determines the operation performed by the integer processing/register file unit 271, and provides an output enable signal. In an exemplary embodiment, the integer processing/register file unit 271 comprises four Integrated Devices Technology IDT49C402A 16-bit bitslices.

Detailed Description Paragraph Right (76):
The multiply/divide unit 276 comprises a 64-bit floating point multiplier/divider having a first input, a second input, a control input, and an output. The control input of the multiply/divide unit 276 is coupled to the second control output of the FU microsequencer 272. The first input of the multiply/divide unit 276 is coupled to the first 64-bit line via a first 32-bit line 282, and the second input of the multiply/divide unit 276 is coupled to the first 64-bit line via a second 32-bit line 283. Therefore, the first input of the multiply/divide unit 276 receives a first 32-bit portion of a 64-bit value, and the second input of the multiply/divide unit 276 receives a second 32-bit portion of a 64-bit value. The multiply/divide unit 276 performs a multiplication or a division upon the first and second 32-bit portions based upon the signal received at its control input. The result of the operation is output at the multiply/divide unit's output. In an exemplary embodiment, the multiply/divide unit 276 is an Analog Devices ADSP3212 64-bit floating point multiplier/divider.

Detailed Description Paragraph Right (77):
The add/subtract unit 274 comprises a 64-bit floating point adder/subtractor having a first input, a second input, a control input, and an output. The control input of the add/subtract unit 274 is coupled to the third control output of the FU microsequencer 272. The first input of the add/subtract unit 274 is coupled to the first 64-bit line via a third 32-bit line 284, and the second input of the add/subtract unit 274 is coupled to the first 64-bit line via a fourth 32-bit line 285. Therefore, the first input of the add/subtract unit 274 receives the first 32-bit portion of a 64-bit value, and the second input of the add/subtract unit 274 receives the second 32-bit portion of a 64-bit value. The add/subtract unit 274 performs an addition or subtraction upon the first and second 32-bit portions based upon a signal received at its control input. The result of the operation is output at the add/subtract unit's output. In an exemplary embodiment, the add/subtract unit 274 is an Analog Devices ADSP3222 64-bit floating point adder/subtractor.

Detailed Description Paragraph Right (78):
The floating point register 278 comprises a 64-bit data storage means having a data input, a data output, and a control input. The control input of the floating point register 278 is coupled via line 288 to the FPR CTRL output of the FU microsequencer 272 to receive control signals. The input of the floating point register 278 is coupled to the output of the multiply/divide unit 276 and to the output of the add/subtract unit 274. The output of the floating point register 278 is coupled to the first 64-bit line 281 via a second 64-bit line 286. The floating point register 278 receives and latches the result generated by the multiply/divide unit 276 or the add/subtract unit 274, and supplies this result to the first 64-bit line via the second 64-bit line 286. By supplying the result back to the first 64-bit line, the result can be stored in the integer processing/register file unit 271, or the result can be operated upon by the multiply/divide unit 276 or the add/subtract unit 274.

Detailed Description Paragraph Right (79):
Referring now to FIG. 9, a flowchart of a preferred method for hyperscalar instruction issue is shown. The preferred method begins in step 1000, with an assignment of an order number to each instruction within the issue consideration group. Each order number indicates the serial arrangement of instructions within the instruction buffer 130. Next, in step 1001, the preferred method determines for each instruction a first index for the first source register file 260. Following step

1001, the preferred method determines for each instruction in step 1002 a second index for the second source register file 260. After step 1002, the preferred method determines for each instruction a third index for the result destination file in step 1003. Following step 1003, the preferred method proceeds to step 1004 and step 1010 simultaneously. In step 1004, the preferred method selects the instruction having the second lowest order number as the current instruction under consideration. Since the instruction having the lowest order number is issued regardless of whether additional instructions can be issued with it in parallel, step 104 does not consider the instruction having the lowest order number. After step 1004, the preferred method determines for the current instruction under consideration whether the first index is equal to the third index. If the first index is not equal to the third index, the first data item required for the execution of the current instruction under consideration is stored in a register file 260 other than the result destination register file 260. This situation corresponds to a register file conflict. In the presence of a register file conflict, the preferred method proceeds to step 1008 and sets a first parallel issue constraint number to the order number of the instruction under consideration minus one. Following step 1008, the preferred method proceeds to step 1030.

Detailed Description Paragraph Right (80):
If in step 1005 the first index is found to be equal to the third index, the preferred method proceeds to step 1006 to test whether the second index is equal to the third index. In a manner similar to that in step 1005, if the second index is not equal to the third index, the second data item required for the execution of the instruction under consideration is stored in a register file 260 other than the result destination file. A register file conflict therefore exists in this situation. If a register file conflict is detected in step 1006, the preferred method proceeds to step 1008. If the preferred method determines that no register file conflict exists in step 1006, operation continues in step 1007 with the preferred method selecting the instruction having the next higher order number as the current instruction under consideration. Following step 1007, the preferred method returns to step 1005.

Detailed Description Paragraph Right (81):
In step 1020, the preferred method uses the instruction having the lowest order number to create a list of previously considered instructions. The instruction having the lowest order number will be issued regardless of whether additional instructions can be issued with it in parallel. Therefore, the instruction having the lowest order number is placed into the list of previously considered instructions by default. Next, in step 1021, the preferred method retrieves the instruction having the next higher order number and identifies it as the current instruction under consideration. The preferred method then determines in step 1022 whether the third index of the current instruction under consideration matches the third index of any instruction in the list of previously considered instructions. A third index match indicates that the instructions corresponding to the match require the same result destination register file 260. Since instructions are issued in parallel, the result destination register file 260 of each instruction issued must be unique to avoid a register file conflict. If no third index matches are found in step 1022, the preferred method proceeds to step 1025 and adds the current instruction under consideration to the list of previously considered instructions. Following step 1025, the preferred method returns to step 1021. If it is determined in step 1022 that the third index of the current instruction matches one or more result destination files in the list of previously considered instructions, the preferred method proceeds to step 1023. In step 1023, the preferred method sets a second parallel issue constraint number to the order number of the current instruction under consideration minus one. After step 1023, the preferred method proceeds to step 1030.

Detailed Description Paragraph Table (1):
APPENDIX A

Hyperscalar Computer Architecture Instruction Set Mnemonic Instruction Description Mixed Radix Code Instruction Operation
AND i,j,k
64-bit bitwise AND 00.vertline.j.vertline.k.vertline.i Ri = Rj & Rk OR i,j,k 64-bit

bitwise OR 01.vertline.j.vertline.k.vertline.i Ri = Rj .vertline. Rk XOR i,j,k 64-bit bitwise XOR 02.vertline.j.vertline.k.vertline.i Ri = Rj Rk INV i,j,k 64-bit bitwise invert (NOT) 03.vertline.000.vertline.k.vertline.i Ri = .about.Rk LSL i,j,k 64-bit logical shift left 04.vertline.j.vertline.k.vertline.i (ulong)Ri = (ulong)Rj << Rk LSR i,j,k 64-bit logical shift right 05.vertline.j.vertline.k.vertline.i (ulong)Ri = (ulong)Rj >> Rk ROL i,j,k 64-bit logical rotate left 06.vertline.j.vertline.k.vertline.i (ulong)Ri = (ulong)Rj rol Rk ROR i,j,k 64-bit logical rotate right 07.vertline.j.vertline.k.vertline.i (ulong)Ri = (ulong)Rj ror Rk BYTE i,j,k right-align selected 8-bit field 08.vertline.j.vertline.k.vertline.i Ri = (Rj >> ((Rk &= 0.times.7) << 3)) &= 0xFF LADD i,j,k 64-bit 2's complement integer add 09.vertline.j.vertline.k.vertline.i (long)Ri = (long)Rj + (long)Rk LSUB i,j,k 64-bit 2's complement integer sub 0A.vertline.j.vertline.k.vertline.i (long)Ri = (long)Rj - (long)Rk ADDQ i,SDATA 18-bit SDATA added to Ri 0B.vertline.SDATA.vertline.i (long)Ri += (long)SDATA ADD i,j,k 64-bit floating-point addition 0C.vertline.j.vertline.k.vertline.i (double)Ri = (double)Rj + (double)Rk SUB i,j,k 64-bit floating-point subtract 0D.vertline.j.vertline.k.vertline.i (double)Ri = (double)Rj - (double)Rk MUL i,j,k 64-bit floating-point multiply 0E.vertline.j.vertline.k.vertline.i (double)Ri = (double)Rj * (double)Rk DIV i,j,k 64-bit floating-point divide 0F.vertline.j.vertline.k.vertline.i (double)Ri = (double)Rj / (double)Rk LD i,*j,K LD 1 to K words into successive Ri 10.vertline.j.vertline.KDATA.vertline.i for (w=0;w 0) 17.vertline.RADDR.vertline.i CIP = ((long)Ri > 0) ? CIP+=RADDR: CIP++ BMI i,RADDR Branch to CIP+=RADDR if (Ri < 0) 18.vertline.RADDR.vertline.i CIP = ((long)Ri < 0) ? CIP+=RADDR: CIP++ BREQ i,*k Branch to *Rk if (Ri == 0) 19.vertline.000.vertline.k.vertline.i CIP = ((long)Ri == 0) ? Rk : CIP++ BRPL i,*k Branch to *Rk if (Ri > 0) 1A.vertline.000.vertline.k.vertline.i CIP = ((long)Ri > 0) ? Rk: CIP++ BRMI i,*k Branch to *Rk if (Ri < 0) 1B.vertline.000.vertline.k.vertline.i CIP = ((long)Ri < 0) ? Rk: CIP++ BIEQ i,UADDR Branch to UADDR+*R0+CIP++ if (Ri == 0) 1C.vertline.UADDR.vertline.i CIP = ((long)Ri == 0) ? (UADDR+*R0) +CIP++: CIP++ BIPL i,UADDR Branch to UADDR+*R0+CIP++ if (Ri > 0) 1D.vertline.UADDR.vertline.i CIP = ((long)Ri > 0) ? (UADDR+*R0) +CIP++: CIP++ BIMI i,UADDR Branch to UADDR+*R0+CIP++ if (Ri < 0) 1E.vertline.UADDR.vertline.i CIP = ((long)Ri < 0) ? (UADDR+*R0) +CIP++: CIP++ L i,j,k microprogram jumpmap via tuple {i,j,k} 1F.vertline.j.vertline.k.vertline.i no state change to M1

Registers

Current US Original Classification (1):
712/24

Current US Cross Reference Classification (1):
712/215

Current US Cross Reference Classification (2):
712/216

Current US Cross Reference Classification (3):
712/23

Current US Cross Reference Classification (4):
712/232

Other Reference Publication (6):
Robert P. Colwell et al, "Architecture and Implementation of a VLIW Supercomputer", IEEE Computer Society Press, Nov. 1990, pp. 910-919.

CLAIMS:

1. In a computer system capable of issuing a plurality of instructions in parallel, a method for determining the number of instructions that can be issued without conflict, the method comprising the steps of:

assigning an order number to each instruction within an issue consideration group;

determining for each instruction a first index for the first source register file of

the instruction;

determining for each instruction a second index for the second source register file of the instruction;

determining for each instruction a third index for the result destination file of the instruction;

comparing the third indices of the instructions in the issue consideration group to determine an instruction at which a destination register file conflict exists comprising the steps of,

a) using a first instruction to create a list of previously considered instructions;

b) retrieving the instruction having the next highest order number as the current instruction under consideration;

c) determining whether the destination register file of the current instruction is the same as the destination register file of any instruction in the list;

d) adding the current instruction to the list and returning to step b) if the destination register file of the current instruction is not the same as the destination register file of any instruction in the list; and

e) outputting the order number of the current instruction under consideration minus one if the destination register file of the current instruction is the same as the destination register file of any instruction in the list; and

comparing the third index to the first and second indices for each instruction in the issue consideration group to determine an instruction at which a source register file conflict exists.

2. In a computer system capable of issuing a plurality of instructions in parallel, a method for determining the number of instructions that can be issued without conflict, the method comprising the steps of:

assigning an order number to each instruction within an issue consideration group;

determining for each instruction a first index for the first source register file of the instruction;

determining for each instruction a second index for the second source register file of the instruction;

determining for each instruction a third index for the result destination file of the instruction;

comparing the third indices of the instructions in the issue consideration group to determine an instruction at which a destination register file conflict exists; and

comparing the third index to the first and second indices for each instruction in the issue consideration group to determine an instruction at which a source register file conflict exists comprising the steps of,

a) selecting the instruction having the second lowest order number as a current instruction under consideration;

b) determining whether the third index for the current instruction is the same as the second index for the current instruction;

c) determining whether the third index for the current instruction is the same as the first index for the current instruction;

d) outputting an instruction order number of the current instruction minus one if

the third index for the current instruction is not the same as the second index for the current instruction;

e) outputting an instruction order number of the current instruction minus one if the third index for the current instruction is not the same as the third index for the current instruction; and

f) selecting the instruction having the next higher order number as the current instruction under consideration if the second index is equal to the third index and the first index.

3. A computer system which issues a plurality of instructions in parallel comprising:

a central memory for storing sequences of said instructions:

a plurality of functional units for processing data in accordance with said sequences of instructions:

a plurality of register files for supplying data to said functional units and for receiving data from said functional units:

an instruction buffer for receiving said sequences of instructions transferred from said memory; and

a control unit which receives an issue consideration group comprising a set of instructions transferred from said memory and which automatically determines a largest subset of instructions within said issue consideration group which can be simultaneously issued to said functional units without conflicts, said control unit comprising;

means for assigning an order number to each instruction within said issue consideration group;

means for determining for each instruction a first index for the first source register file of the instruction;

means for determining for each instruction a second index for the second source register file of the instruction;

means for determining for each instruction a third index for the result destination file of the instruction;

means for comparing the third indices of the instructions in the issue consideration group to determine the instruction at which a destination register file exists comprising,

means for using a first instruction to create a list of previously considered instructions;

means for retrieving the instruction having the next highest order number as the current instruction under consideration;

means for determining whether the destination register file of the current instruction is the same as the destination register file of any instruction in the list;

means for adding the current instruction to the list and invoking said means for retrieving the instruction if the destination register file of the current instruction is not the same as the destination register file of any instruction in the list; and

means for outputting the order number of the current instruction under consideration minus one if the destination register file of the current instruction is the same as the destination register file of any instruction in the list; and

means for comparing the third index to the first and second indices for each instruction in the issue consideration group to determine the instruction at which a source register file conflict exists.

4. A computer system which issues a plurality of instructions in parallel comprising:

a central memory for storing sequences of said instructions:

a plurality of functional units for processing data in accordance with said sequences of instructions:

a plurality of register files for supplying data to said functional units and for receiving data from said functional units:

an instruction buffer for receiving said sequences of instructions transferred from said memory; and

a control unit which receives an issue consideration group comprising a set of instructions transferred from said memory and which automatically determines a largest subset of instructions within said issue consideration group which can be simultaneously issued to said functional units without conflicts, said control unit comprising;

means for assigning an order number to each instruction within said issue consideration group;

means for determining for each instruction a first index for the first source register file of the instruction;

means for determining for each instruction a second index for the second source register file of the instruction;

means for determining for each instruction a third index for the result destination file of the instruction;

means for comparing the third indices of the instructions in the issue consideration group to determine the instruction at which a destination register file exists comprising,

means for comparing the third index to the first and second indices for each instruction in the issue consideration group to determine the instruction at which a source register file conflict exists comprising,

means for selecting the instruction having the second lowest order number as a current instruction under consideration:

means for determining whether the third index for the current instruction is the same as the second index for the current instruction;

means for determining whether the third index for the current instruction is the same as the first index for the current instruction;

means for outputting an instruction order number of the current instruction minus one if the third index for the current instruction is not the same as the second index for the current instruction;

means for outputting an instruction order number of the current instruction minus one if the third index for the current instruction is not the same as the third index for the current instruction; and

means for selecting the instruction having the next higher order number as the current instruction under consideration if the second index is equal to the third index and the first index.